NPS ARCHIVE
1998.12
ZACHARY, U.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MULTIPLE ROBOT COMMAND AND CONTROL ARCHITECTURE DEVELOPMENT

by

Uriah E. Zachary

December 1998

Thesis Advisor: Xiaoping Yun

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
MULTIPLE ROBOT COMMAND AND CONTROL ARCHITECTURE DEVELOPMENT

5. FUNDING NUMBERS

6. AUTHOR(S)
Zachary, Uriah E.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Space and Naval Warfare System Center - San Diego
San Diego, CA 92152-5001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The military use of autonomous vehicles or robots will increase as national security planners seek to maintain strategic deterrence and preserve U.S. interest in spite of reduced resources. Cooperative group behavior among large numbers of robots will be required to complete various missions. Communication schemes for command, control, and coordination of multiple robots is one of the required capabilities. This thesis evaluates the Simplified Lisp-like Expression Evaluation Paradigm (SLEEP) for implementation as a development tool and a communications scheme. SLEEP enables the dynamic group formation of robots that are best qualified for a task. The SLEEP concept is tested and evaluated using a testbed built from Nomadic SCOUT mobile robots and a socket interface. Results from simulation and physical experiments validate the effectiveness of SLEEP for multiple robot coordination.

14. SUBJECT TERMS Autonomous Agents, Multiple Robots, Cooperative Behavior, Dynamic Addressing			15. NUMBER OF PAGES 114
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFI- CATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std.

Approved for public release; distribution is unlimited

MULTIPLE ROBOT COMMAND AND CONTROL ARCHITECTURE DEVELOPMENT

Uriah E. Zachary
Lieutenant, United States Navy
B.S., California State University, Northridge, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1998

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

ABSTRACT

The military use of autonomous vehicles or robots will increase as national security planners seek to maintain strategic deterrence and preserve U.S. interest in spite of reduced resources. Cooperative group behavior among large numbers of robots will be required to complete various missions. Communication schemes for command, control, and coordination of multiple robots is one of the required capabilities. This thesis evaluates the Simplified Lisp-like Expression Evaluation Paradigm (SLEEP) for implementation as a development tool and a communications scheme. SLEEP enables the dynamic group formation of robots that are best qualified for a task. The SLEEP concept is tested and evaluated using a testbed built from Nomadic SCOUT mobile robots and a socket interface. Results from simulation and physical experiments validate the effectiveness of SLEEP for multiple robot coordination.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. GENERAL 1
B. PROBLEM STATEMENT 3
C. OUTLINE OF THE THESIS 4

II. DEVELOPMENT ENVIRONMENT 7

A. HARDWARE DESCRIPTION 7
1. Nomad Scout Mobile Robot 7
2. Ethernet 9
B. SOFTWARE DESCRIPTION 9
1. Nomadic Host Software Development Environment 9

III. SIMPLIFIED LISP-LIKE EVALUATION EXPRESSION PARADIGM 15

A. MOTIVATION 15
B. LANGUAGE ATTRIBUTES 17
C. DEFINITION 19
D. APPLICATIONS 24

IV. SOCKET-BASED INTERPROCESS COMMUNICATIONS 29

A. THE NETWORK MODEL 29
B. THE CLIENT/SERVER MODEL 31
C. TRANSPORT PROVIDERS 32
D. THE SOCKET INTERFACE 33

V. DESIGN APPROACH 37

A. TESTBED CREATION 37
1. Integration 38
B. SLEEP EVALUATION 39

VI. RESULTS 41

A. SLEEP EVALUATION 41
1. Compilation and Portability 41
2. Syntax Determination 41
3. Expression Evaluation 42
B. BROADCAST MODULE 46
C. INTEGRATED PROGRAM TESTING 46
1. Simulation 47
2. Real World 49

VII. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE STUDY .. 53

APPENDIX A. SLEEP.C 55
APPENDIX B. TUNE-UP.C 63
APPENDIX C. SOCKET_READ.C 69
APPENDIX D. SOCKET_SEND.C 71
APPENDIX E. SOCKET_SHELL.C 73
APPENDIX F. SLEEPY.C 75
APPENDIX G. BRDCST.C 83
APPENDIX H. ROB1.C 85
APPENDIX I. ROB2.C 91
APPENDIX J. CODE SEGMENTS FOR LINKING 97
LIST OF REFERENCES 99
INITIAL DISTRIBUTION LIST 101

LIST OF FIGURES

Figure 1. NOMAD SCOUT Mobile Robot.....7
Figure 2(a). GUI: World Map.....12
Figure 2(b). GUI: Robot Map.....12
Figure 3. Sleep Address Space.....21
Figure 4. OSI Reference Model From Ref. 16.....30
Figure 5. Socket Connection From Ref. 16.....34
Figure 6. Datagram Communication After Ref. 16.....35
Figure 7. Socket/Application Functional Flow Diagram.....40
Figure 8. Initial State of Simulation.....48
Figure 9. Final State of Simulation.....48
Figure 10. Real-world Test Environment.....51

ACKNOWLEDGEMENTS

I would like to acknowledge the financial support of the Space and Naval Warfare System Center-San Diego (SPAWARSYSCEN SD) Research Fellowship Program. Because of it, travel and the purchase of necessary equipment were possible.

I would like to thank my thesis advisor, Professor Xiaoping Yun, for his support, patience, guidance and encouragement. Working with him was fun, rewarding, and enriching.

Many thanks to my second reader and SPAWAR sponsor Dr. Douglas W. Gage. It was a pleasure to work with the originator of the SLEEP concept. Our visits always added more insight and enjoyment to the discipline. His final support and responsiveness was a saving grace!

Thanks to John Lock, NPS computer Science Dept., for the introduction to sockets. Without this, the thesis would have taken a different direction.

Thanks to Art Neumann, NPS ECE Controls Laboratory, for his daily humor, support, and teachings for survival in the world of UNIX and Windows.

I would like to acknowledge the love, patience and support of my lovely wife Karen - We did it!

I would like to dedicate this thesis to my parents for they stressed the importance of education and hard work. The early robot toys were not forgotten.

I. INTRODUCTION

A. GENERAL

As the reduction in American military forces continues, caution must be exercised for the assurance of minimal adequate force structure. This presents an immediate challenge to national security planners as it is essential for the maintenance of strategic deterrence and the support of U.S. political and economic objectives abroad. At present, in support of these objectives, military forces continue to do more with less. Battlefields of the future will require less manning because of the proliferation of sensors and emitters. These devices may be realized through micro-technology for the creation of area denial systems, "pop-up" and "fire ant" warfare. This may be one form of network-centric warfare. Battlefields may be saturated with hundreds of sensors and autonomous vehicles, armed with smart munitions that loiter or hibernate until the target or victim traverses the area. This will render obsolete the traditional concept of organizing forces around major platforms since those platforms will be much more vulnerable/outnumbered. Groups of cooperative agents or evolving robots can help fill this gap.[Ref. 1]

Results of a recent military robotics workshop identified increasing needs for military robotics and identified the current deficiencies. Some of the robotic applications likely to be of greatest utility to the

military include: systems capable of switching easily between missions; systems that can operate either in the air or on the ground; micro-robots; tactile and/or kinematic sensing robots; robots with pattern recognition capabilities; robots with rudimentary abilities to solve problems, address novel situations, reason, and learn; groups of robots with coordinated, emergent behaviors; robots capable of understanding and responding to human facial expressions; and robots realized through biotechnology. The process of data fusion can be resource-intensive. Robotic systems and technologies may facilitate the merging of all information-related tasks (intelligence gathering, surveillance, target acquisition, information warfare, etc.) into a single "information meta-task". Current robots are "extremely dumb", lacking even insect intellect and thus falling short of the ability to emulate human behavior and intelligence. Communication links between operator and robot as well as between multiple robots need to be improved. Better communication sub-systems are required for integration, command, and control of large groups of robotics systems in attempt to benefit from emergent or cooperative robotic behaviors. Other nations or adversaries will compete with the United States in the development of robotic capabilities. Accordingly, design, developmental testing and procurement issues must be addressed.[Ref. 2]

B. PROBLEM STATEMENT

The realization of task-driven cooperative behavior among a group of autonomous agents is formidable because of the required coordination. The same problem plagues other social orders such as urban areas, insect colonies, animal herds, etc. The members may desire individual rights and freedom of choice yet the society may perish if the members don't collaborate for achieving beneficial goals for the society. Communication among the members, and with any external ruling or intruding bodies, must be possible for the occurrence of cooperation. Determination of a feasible communications scheme requires the use of development and diagnostic tools to capture and evaluate the reactive behavior.

Development of a many-robot system may present an agonizing process of testing and debugging. The operating environment as perceived by the robots' sensor suite may not match the environment as perceived by the human developer. Accordingly, the reactive behavior of the robots may not match the intentions of the developer. Furthermore, acceptable behavior in one scenario may be chaotic in another scenario without any external or simulated indications of the cause. Determination of such causes requires a communications scheme that provides controllability and observability of the robots' states. One proposal is the use of a high bandwidth broadcast

channel which transmits messages from the user's workstation to the robots. This thesis attempts to evaluate and implement the SLEEP (Simplified Lisp-like Expression Evaluation Paradigm) command channel/broadcast scheme as a communications scheme and diagnostic tool for multiple-robot command and control (C2) architecture development. SLEEP proposes a means to address a message to a specific individual robot or group of robots as the number of robots grows arbitrarily large. Each robot determines whether a given command is intended for it by evaluating a predicate expression based on the values of its own state variables. [Ref. 3]

The integration of existing resources will be used for SLEEP implementation and evaluation. The introduction of an interprocess communication scheme, based on network protocols, in a robot software development environment will comprise the SLEEP testbed. Success in this endeavor may enable experimentation in reactive group behavior which is a necessity for the future robotic battlefield and commercial arena.

C. OUTLINE OF THE THESIS

Chapter II provides a description of the platforms, software and hardware used to conduct the research. Chapter III discusses the SLEEP concept and its motivation and advantages while chapter IV examines the background, issues and current uses of sockets, network protocols, interprocess

communications and alternatives. Chapter V describes the methods pursued for multiple robot integration/development. Chapter VI presents the integration, test, and evaluation results of SLEEP implementation. Chapter VII presents conclusions and recommendations for future study.

II. DEVELOPMENT ENVIRONMENT

A. HARDWARE DESCRIPTION

1. Nomad Scout Mobile Robot

The NOMAD SCOUT is an integrated mobile robot system with ultrasonic and tactile sensing modules. It uses a hierarchial control architecture. A special multiprocesssor-based low level control system performs sensor, motor, and communication process control. High level control is provided by either a UNIX-based, laptop computer or a remote workstation. SUN Sparc workstation were used for this study. The NOMAD SCOUT is software compatible with the NOMAD 200 class robots, another research platform at NPS [Ref. 4]. Figure 1 shows a picture of the NOMAD SCOUT.

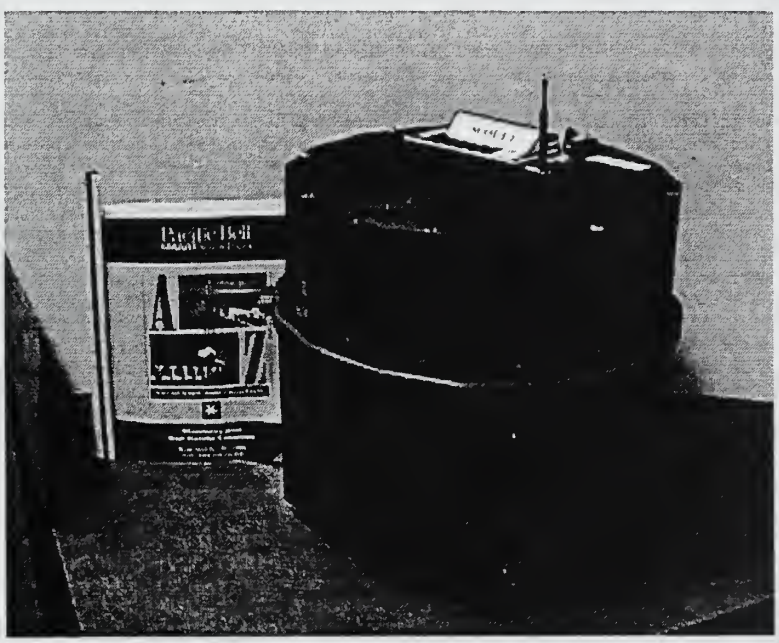

Figure 1. NOMAD SCOUT Mobile Robot

a) Mechanical System

The NOMAD SCOUT is a two-degree of freedom (DOF) differential drive robot with omnidirectional motion. Translation and steering (or rotational motion) are not decoupled since the robot's differential drive base has axes that serve as both translation and steering. The drive is set about the geometrical center of the robot. Compact design of the SCOUT made it an attractive choice for multi-robot experimentation. It has a height of 0.38 meters, a diameter of 0.34 meters and a 3.5 centimeter ground clearance. Without batteries, the unit has a mass of 23 kilograms. Maximum speed and acceleration are 1.0 meters per second and two meters per second squared, respectively. [Ref. 5]

b) Sensor System

The NOMAD SCOUT perceives its environment through the use of ultrasonic, tactile (bumper) and odometric sensor systems. The ultrasonic sensor system uses 16 independent standard Polaroid transducers. The effective range of the ultrasonic sensors is 6 to 255 inches. This sonar system is practically identical to the one installed in the NOMAD 200 with slight differences due to the smaller diameter of the NOMAD SCOUT [Refs. 4,5,6]. The tactile system uses a ribbon switch enclosed in a energy absorbing neoprene channel to provide 360-degree coverage. The odometric

sensors are provided on the base of the SCOUT in the form of incremental angle encoders which enable movement tracking. The encoder resolution is 167 counts/cm for translation and 45 counts/degree for robot rotation. [Ref. 7]

2. Ethernet

Communication with and/or among robots occurs over a radio-ethernet that uses a RangeLan2/Access Point bridge device. Each robot has a 2.4 Ghz radio modem and a corresponding IP address for the reception of instructions. Carrier Sense Multiple Access/collision Avoidance (CSMA/CD) is the governing media access protocol. [Ref. 8]

B. SOFTWARE DESCRIPTION

1. Nomadic Host Software Development Environment

The NOMAD SCOUT can be programmed using the Unix-based Nomadic Software Development Environment. It is a full featured, object based, integrated package which consists of two parts: the server (host workstation) and the client (robot). The server is a convenient way to send commands to the robot and receive sensing data from the robot. The server is run as a separate process on a workstation. This process communicates with the robot process (client) through the radio ethernet using the TCP/IP protocol. The server performs four functions: Host <-> Robot Interface for complete control of the robot from a host computer, a graphic user interface which provides a graphic display of

sensor information and convenient interface with the robot and robot simulator, a simulator and a Client <-> Server Language User interface which allows a C or Lisp application program (which acts as a client process) to access the server (Nserver). [Ref. 4]

a) Host Robot Interface

Programs running on-board the robot go into a loop, assimilate sensory data, and wait for commands from the host computer. The robot responds to the set of predefined commands. To command the robot, the host computer sends a stream of characters via the radio modem to the robot and waits for the robot's reply. The robot's reply consists of a stream of characters. A simple synchronous serial communication protocol facilitates the communication between the host computer (server) and the robot (client). These protocols are used to implement motion and sensor commands. The motion commands allow the user to set the speed and acceleration of the robot and to command the robot to move at certain velocities or to move by a certain distance. The sensor commands allow the user to configure and acquire data from the sonar and bumper sensor systems. [Ref 4]

b) Graphical User Interface

The Graphical User Interface (GUI) provides consolidated access to the real and simulated robots and to

their world representation as shown in figures 2(a) and 2(b). Through the GUI, the user can send commands to the robots, monitor command execution by seeing the actual robot motion on a screen window, and visualize real-time and historical sensor data. The user can create and modify a simulated environment for testing robot programs. The graphic environment consist of four main windows: the World or Map window, the Robot window, the ShortSensors window and the LongSensors window. The World window represents the environment from a frame of reference or rectangular coordinate system common to all robots (i.e. the world window, global view). This window allows for manipulation of the environment through the creation of simulated obstacles. The Robot window displays the world as seen by one robot (local view). This window allows the display of sensor history, robot path, as well as the execution of robot commands. Both reference windows support display functionalities like zooming, scrolling, centering, etc. The ShortSensor window provides an optional display of obstacle detection by infrared and/or bumper hit/collision.

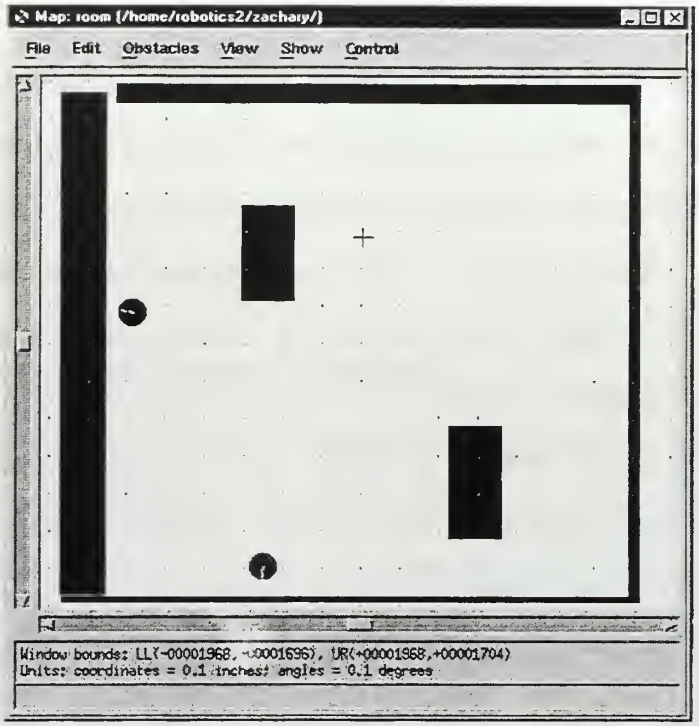
Map: room (/home/robotics2/zachary/)
File Edit Obstacles View Show Control
Window bounds: LL(-00001968,-00001696), UR(+00001968,+00001704)
Units: coordinates = 0.1 inches; angles = 0.1 degrees

Figure 2(a). GUI: World Map

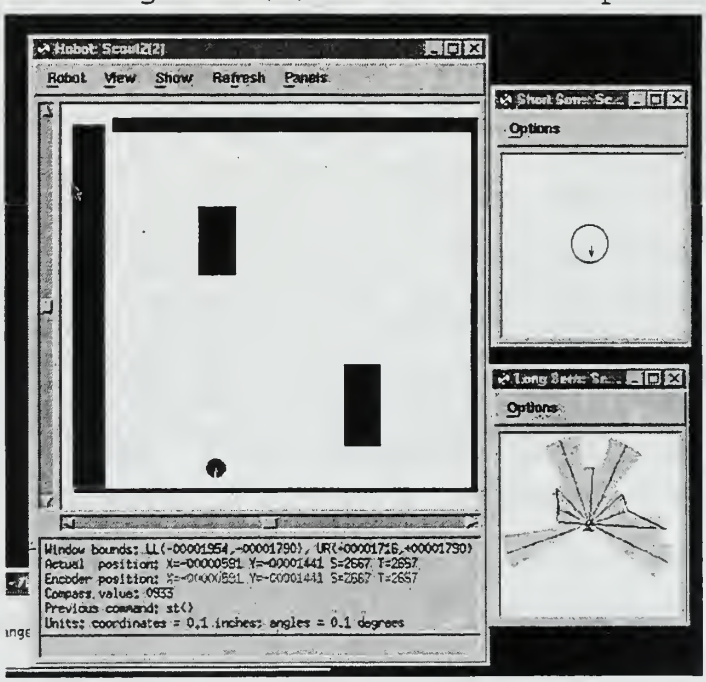

Figure 2(b). GUI: Robot Map

Infrared sensors are not available on the SCOUT. The LongSensor window allows the display of Laser or sonar

sensor data. The laser is not available for the NOMAD SCOUT.

c) Robot Simulator

The simulator models the robot's basic motion (translation and steering) and its sensor systems. The robot can only translate along the forward and backward directions in which the wheels are aligned. This is often referred to as a non-holonomic constraint similar to that of a car. Unlike a car, the robot can rotate in place (zero turning-radius). To model uncertainty, the simulator keeps track of two positions of the simulated robot: encoder and actual position. This provides representation of the commanded (desired) velocity or position and the actual effects of slippage or the sliding of wheels. The randomness factor of the sensed distance for sonar is also taken into account by the simulator. A control uncertainty model and sonar simulation parameters are specified in the robot.setup file and can be adjusted for better modeling in a given environment. The simulated robot responds to the same sets of commands as the real robot. There is no limit to the number of real or simulated robots that can be hosted; however, network congestion/response time may become a factor when exceeding seven robots.

d) Language User Interface

The Language User Interface provides the link between the application program (as a client process) and the robot or robot simulator and the GUI (as the server process). For C language interface, one can include Nclient.h in the application program and link it to Nclient.o. If the application program is not running on the same computer where the server is running, then the host computer must be designated as the server machine in the application program. The TCP port number used by the server (as specified in the world.setup file) must be the same as that specified in Nclient.o and the application program. This is required for the server and application to communicate. There are two global variables defined in the file Nclient.h which the application program can access and there are a set of interface functions which the program can call. The state vector global variable is an array of 45 long integers which reflect the current state of the robot. The Smask global variable specifies which sensor data are to be returned and hence which sensor data is to be disregarded, not processed or "masked".

III. SIMPLIFIED LISP-LIKE EVALUATION EXPRESSION PARADIGM

A. MOTIVATION

Unambiguous communications among any group is required for cooperative behavior in support of unified, group-oriented task achievement. For member or group survival in a dynamic environment, the communications scheme or language must facilitate information flow that will enable adaptation to the current environment. In Robotics and AI, total reliance upon "hard-wired" or preprogrammed instructions do not provide such flexibility. A group is only as efficient as it can be efficiently organized. For example, if a group consist of, say, one thousand robots with the same application program and the environment or tactical situation unexpectedly changes, then how does the group reconfigure itself? Intervention would be required from the system developer or mission commander. All one thousand robots would have to be stopped, taken off-line or out of operation and another program or modification would have to be downloaded. How long would this take!? What implication or impact would this have on the now-suspended operation (i.e. an assembly line, construction site, tactical engagement). What would be the subsequent level of productivity or mission readiness? In military operations, organization is based on the chain-of-command where key information or directives flow from National Command Authority down to the deployed troops. The rules of

engagement define or map defensive and offensive responses to various situations. In the US Navy, if shots are fired and/or communications are lost between the warfare commanders and their immediate superior in command (Officer in Tactical Command), then the warfare commanders can take necessary action without requesting and receiving permission. This offers some level of autonomy without having to rely solely on directives from the top. Here, the organization has a fighting chance because of its ability to react and adapt to scenarios which deviate from the script. Naval force organization evolves around the aircraft carrier and the amphibious command ship. This comprises two groups: the Carrier Battle Group (CVBG) and the Amphibious Readiness Group (ARG). These task forces are further broken into smaller groups such as Task Groups and Task Elements. Protective screens are set around the high value unit by explicitly assigning units to a group or sector. The assignments are not made at random yet conditions may have changed since the assignments were made. They are usually based on the mission, equipment capabilities and personnel. If group assignments could be made instantaneously based on who meets conditions/criteria imposed by the new, emergent task (i.e. downed aircraft, man overboard -- search and rescue; hostile submarine contact and prosecution), then resource management could be optimized and the response time would be reduced. Once again, re-emphasising Chapter I,

multi-tasking could cause us to spread our forces too thin if we do not fully utilize the technology of Robotics and Automation!

Athletics also provides examples of the need for dynamic group reconfiguration ability. In basketball, which defense is best: zone or man-to-man? It may very well depend on the environment, the opponent, injuries, if we are winning or losing and how much time is left in the game. In football, the type of organization employed may be one form if we are approaching half-time and up by five points or another form if we are down by a field goal with two minutes left in the game. This may drive us to use the no-huddle offense unless there is too much noise (i.e. jamming) and the offensive line can't hear the audible or the play being called by the quarterback. In short, if groups are formed by the use of conditionals, qualifiers or thresholds and not explicitly by mechanical unit/name-to-group assignments, then one would achieve a system response or effect more commensurate to the situation. Similar, desirable group behavioral characteristics and the underlying, required developmental process are the motivation for consideration of SLEEP implementation.

B. LANGUAGE ATTRIBUTES

The development of robots with intelligent, evolutionary behavior requires a computer programming language that is designed, itself, to evolve. This is one

of the distinctive characteristics of the LISP programming language. One can use LISP to define new LISP operators. As new abstractions such as object-oriented programming continue to increase in popularity, it turns out easier to implement them in LISP. LISP lets one do things that can't be done in other languages. For example, suppose you want to write a function that takes a number n and returns a function that adds n to its argument. This can be done in LISP but not in C. The phrase rapid prototyping describes a kind of programming that began with LISP; A prototype can be written in less time than it would take to write the specification for one. Such a prototype can be so abstract that it makes a better specification than one written in English. LISP allows for a smooth transition from prototype to production software. [Ref. 9]

LISP takes its name from List Programming. Let's see why. Binary digits (bits) in a computer, taken in groups, can be interpreted as a code for word-like objects and sentence-like objects. In LISP, the fundamental objects formed from bits are word-like objects called atoms. Groups of atoms form sentence-like objects called lists. List themselves can be grouped together to form higher-level lists. Atoms and list collectively are called symbolic expressions or, for short, expressions. Working with symbolic expressions is what symbol manipulation using LISP is about.

Just as people use pencil, paper and human language to remember and work with data and procedures, a symbol manipulation program uses symbolic expressions to remember and work with data and procedures. Typically, the procedures can recognize particular symbolic expressions, tear old ones apart and assemble new ones. In short, symbol manipulation enables LISP to performs functions or task such as expert problem solving, commonsense reasoning, learning, natural language interfaces, education and intelligent support systems and speech/vision synthesis. These language attributes can be used to foster intelligent robot behavior. [Ref. 9]

C. DEFINITION

SLEEP is a communications scheme for addressing a message to a specific individual robot or group of robots as the number of robots grows arbitrarily large. Each robot determines whether a given command is intended for him by evaluating a predicate expression based on the values of his own state variables or, in the case of our test subject, NOMAD SCOUT, state vector. A predicate is an expression whose value is true or false. A predicate can also be thought of as a procedure that returns a value that signals true or false. Consider the following example. A disabled Naval vessel X is in imminent danger of hostile fire. The only friendly forces in the vicinity are robotic scouting crafts that have been dispatched from a "mother ship". They

were widely dispersed for various missions. The compact design of the scouts and their primary mission limits their fuel and ordnance capacity. Which scouts are best qualified to defend or assist the disabled vessel? By broadcasting a message which says "All scouts within 5 miles of X that have more than 50% fuel level and 70% ordnance are now in Task Element Bravo", criteria has been established and disseminated for determining which scouts should assist vessel X. Subsequent commands can be given to the previously selected group. For example, "Task Element Bravo, cover vessel X" or "Task Element Bravo, render rescue and assistance to vessel X" are commands that can be given without knowing the identity (i.e. hull number) of each scout. Through one broadcast command, an organizational category was created, a degree of readiness was indirectly assessed, and an order was given for execution. A LISP rule-like paradigm can be used to realize such a broadcast: IF <predicate> returns true, THEN store and execute <command>. By adopting a LISP representation for <predicate>, we can also use it for the <command>. Thus the IF-THEN rule can be explicitly captured in a LISP COND function and the whole message becomes a single LISP expression whose evaluation is all the processing that is required. [Ref. 3]

SLEEP consists of several address spaces with the associated storage classes and processing: (1) expressions,

(2) variables, and (3) functions as depicted in Figure 3. Each node corresponds to a robot and incorporates an indexed set of buffers for storing a number of LISP-syntax expressions. [Ref. 3]

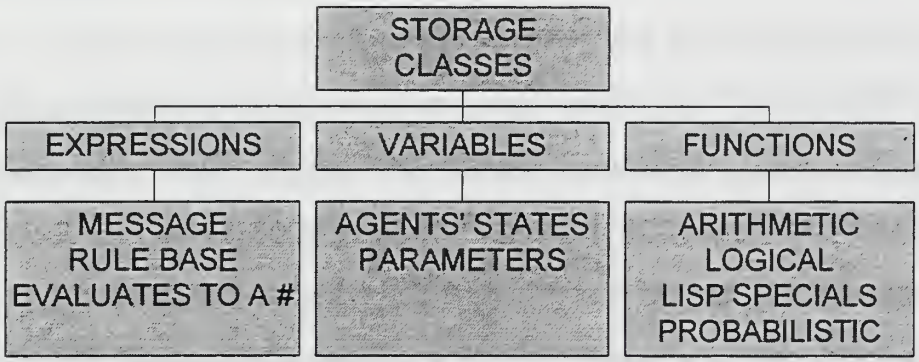

Figure 3. Sleep Address Space

Each expression buffer is either active or inactive by either SETQing the active flag variable or through an explicit activate/deactivate function call. The core SLEEP process evaluates the expressions in the active buffers in a round robin sequence although other priority schemes could be used. The message reception process writes an incoming expression into a special buffer and SLEEP executes it one time. The expression can copy a portion of itself into one of the other regular buffers for repeated processing as a part of the rule base. [Ref. 3]

SLEEP provides an indexed set of variables for potential use in three ways. Some of the indexed variables point to the other variables that are used by the robot's

underlying processing such as memory-mapped sensor inputs and actuator outputs. Others support SLEEP processing itself, like marked, marking, frozen, and sleeping. These proposed processes are not the focus of this thesis and are not addressed. The user has flexibility in using the rest of the variables as deemed necessary. [Ref. 3]

The function base can include basic arithmetic (+, -, *, /, ^, etc), boolean (AND, OR, NOT), LISP "special forms" (COND, SETQ), SLEEP support (power down), and pointers to functions in the robots core software. Randomization functions can include (PROB x), which evaluates as TRUE with probability x, and the special form (DICE (x A)(y B)(z C)), which evaluates A with probability x, B with probability y, and C with probability z. [Ref. 3]

The preliminary, exploratory C-based implementation of SLEEP is given in Appendix A. In this version, the simplified range of LISP expressions, whether in a message or in the rule base, is represented as a sequence of tokens. In the rule base, i.e. varStr[], expStr[], funStr arrays, each token represents a number, a variable name, an expression name, or a function name. These are used to construct SLEEP expressions which are stored internally as an array of shorts. The assigned respective ranges are as follow: -65535 to +29999, 30000 to 30999, 31000 to 31999, and 32000 to 32999. The last three ranges are arbitrarily defined by VARBASE, EXPBASE, and FUNBASE, respectively. The

end of expression delimiter DONE is defined as 32100. Currently, numbers have been limited to integers only but floating point numbers would be required for representing probabilities. The present set of functions implemented include arithmetic, logical, LISP special forms and a variant of SETQ called SETQQ which QUOTEs both of its arguments. [Ref. 10]

SLEEP functions as a LISP interpreter by realizing the effect of LISP-like expressions in a C language environment. The main function of the SLEEP program allows the user to type in a string of characters and end with a carriage return. It then echoes the original string, parses it into an array of tokens which it stores in ex[0], displays the string representing the array of tokens, evaluates it, and displays the results. Since an expression always evaluates to a number, this is not really LISP but encapsulates some desirable features of LISP such as call-by-value parameters, operator prefix notation, and function side-effects. To make this happen, an expression can assume ONLY one of the following forms: (1) numerical-value, (2) defined-variable-name, or (3) (defined-function-name expression1 expression-N). Evaluation of each form will return, respectively, the numerical value, the current value of the variable, or the value of each expression followed by the computed value after the function is applied. [Ref. 10]

D. APPLICATIONS

The SLEEP broadcast channel can serve as a diagnostic, development tool and as a medium for operational deployment of a group of autonomous vehicles. Three recent similar methods were presented at IROS '98 [Ref. 11,12,13]. The first method consisted of a dynamically extendable (i.e. while robots are performing their missions) language which supports nine conversation types and works with 26 message types. The Addressees Identity message field defines agents that are not necessarily concerned by the message. They are potential addressees. When a flag is set, a condition on the agents is included in an extension field. An agent becomes a real addressee when its condition matches the one in the extension field. One agent can control another agent through recruitment orders. The requester sends a broadcast with a list of conditions which must be fulfilled for team membership. Applications for this strategy include exploration or mapping and convoying (leader and pursuer) which is simple operation through synchronization. The second method proposed an algorithm for cooperation among robots. The algorithm provides the capability for each robot to select an appropriate behavior depending on the situation. An operator provides commands to the group and each robot uses energy consumption as one factor for behavior selection. Various desirable behaviors are prepared in advance and included in a behavior module for

robot selection. Cooperative behavior such as task-sharing is achieved by each robot selecting its own behavior while considering the other robots' behavior. The third method and application consist of a robot society used to remove and regulate algae growth, in pipes, by releasing poisonous chemicals. The behavior of the robots is based on selected strategy. There can exist as many groups as there are strategies. The approach attempts to optimize the size of the society for task performance since the performance of the society changes as the size of the society is varied. All strategies are tested where the final chosen strategy returns the greatest profit. While the communications scheme is different, the grouping concept was interesting and similar to concepts investigated in this research.

One given illustrative example of SLEEP application is the consideration of how it might be applied to Hoskins' "emergent Braitenberg Vehicle" [Ref. 14]. This vehicle serves as a model for relating the Principle of Least Action to collective behavior. The Principle of Least Action is rooted in Calculus of Variation for it states that there is a function (action) whose integral is minimized along the path that a particle actually takes as compared to all other possible paths. Conditions of the environment determine the path and thus the behavior of the particle or the vehicle in our case. Hoskins's summary of the behavior is as follows:

"individual behaviors are randomly selected with probabilities determined by the environment".

Hoskins' system consists of dozens of identical mobile vehicles, operating in a plane, that travel at the same constant speed in straight line paths and occasionally "tumble" or turn at random to the left or right by about 67 degrees. Each vehicle can generate a ping or an active transmission or source to indicate its existence as well as detect the ping of other vehicles. The detector can determine the range and relative position of the source or ping associated with the other vehicles. A second sensor can determine the range and relative position of a light source located in the plane. Hoskins divides the vehicles into groups or caste whose caste number is encoded in the ping. Each caste can be configured with a set of simple behavioral rules so that the group as a whole manifests the key characteristics of a classic Braitenberg Vehicle homing toward the light source, including body structure, propulsion, control, and sensing. [Ref. 14]

Behavior is generated by events that depend upon the sensor outputs. The vehicle, robot, or agent compares the sensor output with the event definitions to generate a boolean output. A simple event, E, consists of 4 elements:

$$E = (\{F, R\}, \{\text{caste \#}, S\}, R, \{N, F\}).$$

The first element indicates the location (Front or Rear), relative to the agent's heading, of the on-board

sensor that has detected a ping or light source. The second element indicates the origin of the ping (caste number of the pinging agent or just "S" for (light) source). The third element specifies a nominal range or radius R while the fourth element indicates whether the pinger is Near or Far relative to the nominal range. For example, the definition:

E1 = ((R, 2, 0.0, F), (F, S, 0.0, F))

defines two single events which must be true for the compound event, E1, to be satisfied. If evaluation of E1 returns true then a ping from a caste 2 agent was detected in the rearward sensor at any range greater than zero while the light source was detected by the front sensor at any range greater than zero. Now, resultant behaviors can be cataloged as a function of an event, an action, and a probability. For example, the definition

B2 = {E2, Tumble, 0.50}

with E2 = (F, 1, 0.0, F) specifies that the agent should tumble with a 50% probability when the specified event is true. Here, detection of a ping from a caste 1 agent by the forward sensor is the triggering condition or qualifier. [Ref. 14]

Gage shows how SLEEP is well matched to the behavioral rule structure developed by Hoskins. Let the variables pRange, pBearing, pCaste, and pFlag represent a ping where pFlag is used for synchronizing the processing of pings as

discrete events. Lower level software queues up multiple pings and presents these events one at a time, setting the pFlag. Upon completion of the event, the pFlag is reset. A similar designation may be used for the light source (i.e. sRange, etc.). A typical rule based on Hoskins' behavior definitions would look like the following:

(COND ((AND pFlag (EQ pCaste3) (EQ bBearing FRONT) sFlag (EQ sBearing FRONT) (PROB 0.9)) (PING myCaste));

(COND ((EQ myCaste 4) (SETQ expr5 (QUOTE the-above-expression-to-be-stored)) (SETQ active5 TRUE))).

The second rule is broadcasted by an operator or system developer. The receiving elements of caste 4 would store the first rule in expression buffer 5 and mark this buffer as active. The first rule sets the behavior for any qualifying agent of caste 4. If a caste 4 agent detects a ping on its forward sensor from a caste 3 agent and the light source on its forward light-detecting sensor then ping. Here, ping is the resultant action or response. It could also be specified as TUMBLE. [Ref. 3]

Some examples have been given for uses of SLEEP. Hoskins' event and behavior definitions provide tremendous potential and flexibility for development of a rule base for cooperative behavior and systems analysis. The use of LISP syntax reduces overhead since the message reception processing mechanism also executes any active messages or expressions in the rule base [Ref. 3].

IV. SOCKET-BASED INTERPROCESS COMMUNICATIONS

The merge of computer science and data communications has led to the computer-communications revolution. Accordingly, the previous fundamental differences between the two fields has diminished [Ref. 16]. There is no fundamental difference between data processing (computers) and data communications (transmission and switching equipment). Considerable overlap of the computer and communications industries, from component fabrication to system integration, has occurred. Technology and technical standards organizations are driving toward a single public system that provides uniform access to information sources world-wide. Data, voice, and video communications have become similar processes. The distinction between the single-processor computer, multi-processor computer, local network, metropolitan network, and the long-haul network has become blurred. Emergence of the Internet and networking allows users to capitalize or exploit the overlap of computer and communication systems. This chapter provides an overview of network architecture for use as a foundation for multi-robot control architecture development.

A. THE NETWORK MODEL

Networking is essential in today's computer environment for it turns isolated computers into integrated systems where resources are shared and capacity problems are reduced [Ref. 16]. Most networks are divided into layers that

perform specific functions. Figure 4 shows the International Organization for Standardization (OSI) Reference Model. The layer architecture provides users with the ease of replacing functionality by layer replacement without affecting surrounding layers. Each layer defines a protocol for use between a local and a remote machine. A protocol is a set of rules that the local and remote machine must follow if they wish to communicate with each other. [Ref. 16]

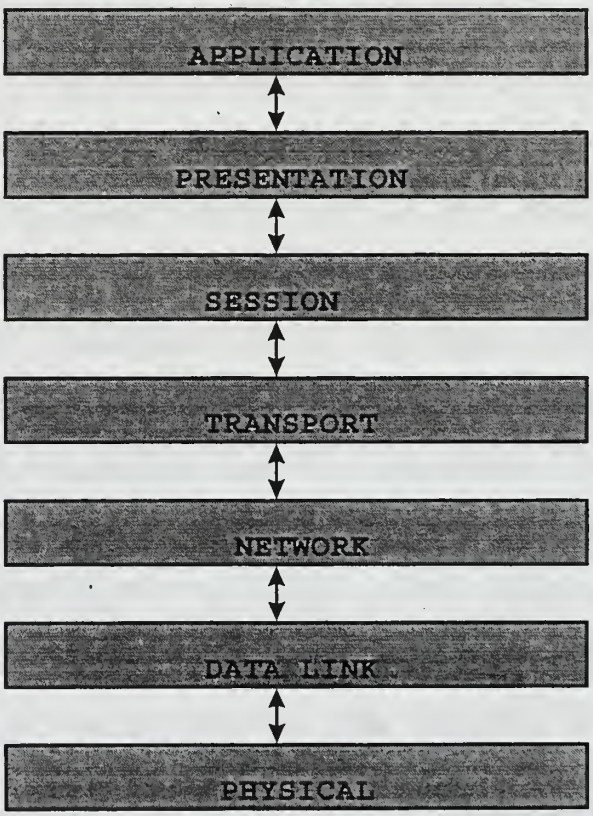

Figure 4. OSI Reference Model From Ref. 16

Opponents to layering believe that better performance can be gained by building an Interprocess Communications

(IPC) facility into a distributed operating system which is supported by software which resides on each host [Ref. 16]. Most multitasking or multiprocessing systems provide IPC facilities so that processes can communicate with each other. Communicating processes which simultaneously run on several processors or share one processor are Concurrent processes. A distributed system is a collection of processing elements which are interconnected, controlled by a system-wide resource manager, and able to execute application processes in a coordinated manner [Ref. 17]. A distributed process can be formed in one of three ways: peer-to-peer, filter, or client/server. The client/server model is the most commonly used paradigm in constructing distributed applications [Ref. 18].

B. THE CLIENT/SERVER MODEL

Most networking applications involve communication between a requester and a supplier of an action or service. This relationship is referred to as the client/server model. A service is a task performed for the requester or client by another machine or server. Communications between the client and server can not occur until a transport address has been established. The server must get the address for the service. As a phone number must be assigned before the user can dial it, so must the server associate the service with an address before clients can communicate with it. The

client must specify the destination address when sending a message to a server. [Ref. 16]

C. TRANSPORT PROVIDERS

Network applications rely on transport providers to transmit data between machines. Transport providers perform the function of the OSI reference transport layer. By accepting network messages and sending them to remote machines, the transport layer frees the application process of the details of achieving data transfer. Transport providers offer two modes of service: connection-oriented and connectionless services. Connection-oriented service is often compared to the telephone call since a phone number must be dialed (by the client) before data can be sent to the server. All messages are gauranteed to arrive in the order in which they were sent. This service is also called a virtual circuit since it resembles a connected circuit between two parties. The connectionless service is similar to sending a message via the post office. Each message must be addressed before it is sent. Messages are put in the mailbox hence the user does not need to know the destination address. While there is a reasonable chance for the message to reach the destination, the sender receives no acknowledgment of message receipt. Furthermore, there is no guarantee that messages will be received in the order in which they were sent. This service is also called the datagram because of its similiarity to telegram service.

The mail service scheme is applicable to multi-robot communications since it is a client/server model of asynchronous communications [Ref. 17]. This represents a general class of IPC models which use multiple processors that share memory. This involves significant operating system overhead since messages must be stored, recipients must be notified, lists of unread messages must be updated, and undelivered mail must be archived. TCP/IP is better equipped for providing such services.

The User Datagram Protocol (UDP) is a transport-level protocol that is in common use as a part of the TCP/IP protocol suite [Ref. 15]. UDP provides the connectionless, primary mechanism that application programs use to send datagrams to other application programs. UDP can distinguish between multiple programs running on the same computer. UDP sits on top of the IP layer, accepts and demultiplexes the incoming datagram based on the UDP destination port. Although UDP is an unreliable service because of unguaranteed message delivery, the overhead of the protocol is low and may be adequate in many cases. [Ref. 19]

D. THE SOCKET INTERFACE

The socket interface is a set of programming subroutines that are used to create a communication channel between resident applications of remote and local systems. Sockets has allowed programmers to use TCP/IP protocols with

little effort [Ref. 19]. Accordingly, the developer can fabricate an IPC facility with little effort. As shown in Figure 4, the socket is an endpoint of a two-way communications channel that can be connection-oriented or connectionless as previously discussed. Figure 5 shows some of the socket routines that let user applications interface with the transport layer so that users do not have to worry about the network transmission details.

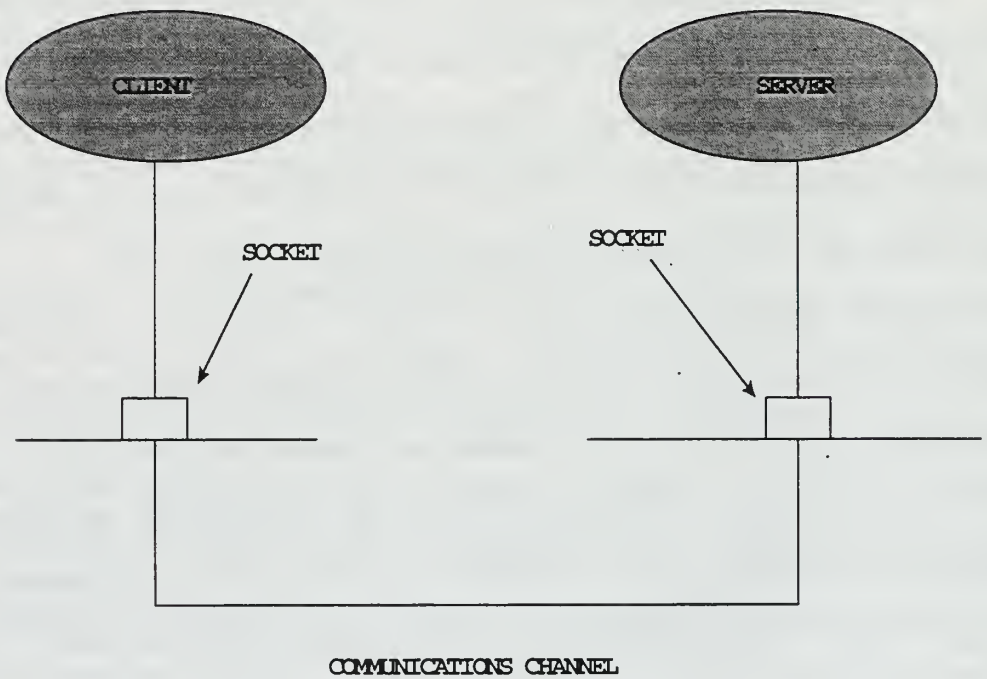

Figure 5. Socket Connection From Ref. 16

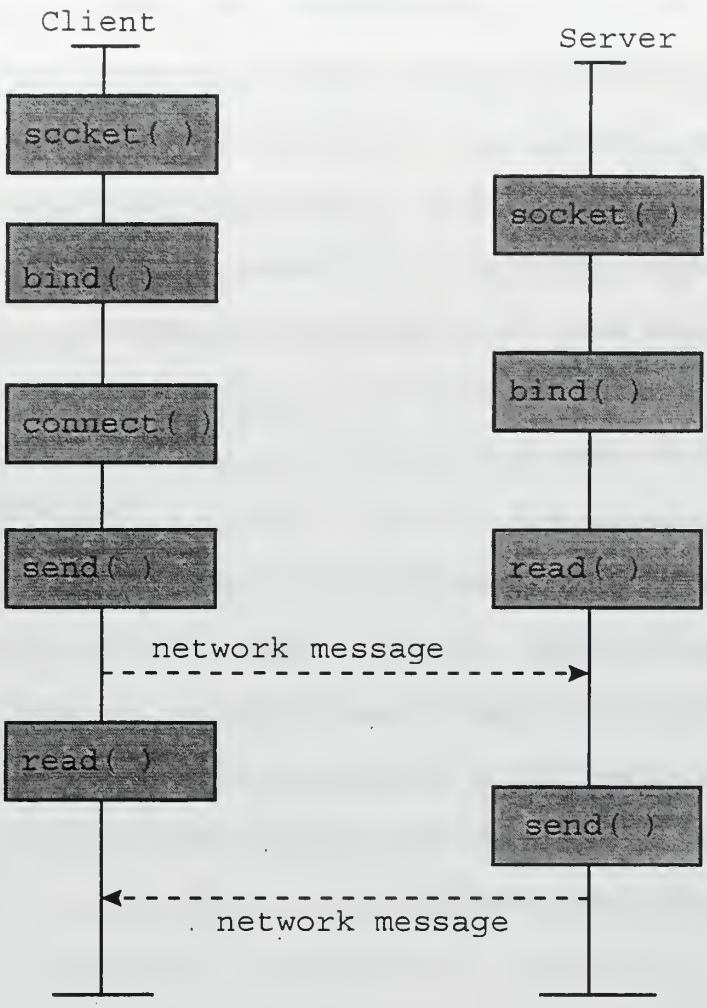

Figure 6. Datagram Communication After Ref. 16

Some datagram transport providers support a transport-level broadcast facility for sending messages to all machines in a network [Ref. 16]. Datagram sockets model contemporary packet switched networks such as the ethernet [Ref. 18]. The socket interface can be used to implement SLEEP because of the IPC facility that it can establish, among agents and developers, across the local ethernet.

Recent studies have been conducted in the area of Web-based communication and control for multi-agent Robots. Focus was placed upon the following Web technologies: browsers, Java language, and socket communication. The relationship between the browser and the sockets was interesting and illuminates the potential expansion of this thesis topic. The browser was proposed as the front-end system for robot control to realize intelligent cooperation between humans and robots. TCP/IP is used for communication from the browser (user) to the agent while UDP/IP is used for communications from the agent to the browser. This mixture places higher communications integrity upon the command transmission than upon the robot reports. This trade-off or balance between integrity and overhead may be suitable for various missions. [Ref. 20]

V. DESIGN APPROACH

Control systems and software design can be best achieved through a modular approach. By decomposing the overarching system function or goal into supporting functions, it may be easier to design and test the system as a sum of its parts. A multi-robot control architecture can be developed in the same manner.

A. TESTBED CREATION

A testbed is required for the development of a multi-robot command and control architecture. This will provide the means for "getting inside the heads of our robots" for determination of their behavior motives. The testbed provides a controlled environment for the examination of cause and effect, or stimulus and response, and the correlation between them. The testbed is created by integrating some aspects of the sockets IPC facility with the existing Nomadic Host Software Development Environment. This provides the ability for a developer to execute an application program/process and modify the process while it is running. By being able to modify the operating parameters while in a dynamic state, the developer debugs the program by making adjustments to achieve the desired result. The testbed may also serve as the operational broadcast command node. These attributes will minimize the number of times the system must be taken off-line for a new mission up-load.

The testbed resides in the Nomadic Host Software Development Environment and consists of an application program, a socket-read module, and a socket-send module, Appendices B, C, and D, respectively. The application program, Tune_up.c is a basic collision avoidance program that demonstrates the basic client interface to the Nserver as well as the reading of sonars and simple motion control [Ref. 21]. The socket modules are basic tutorial examples [Ref. 18].

1. Integration

The integration phase consisted of preliminary test and implementation of the socket modules for interoperability followed by the merge of the socket-read module with the application program. Two SUN Sparc workstations were used as host computers for the socket modules. The interoperability test was conducted by defining a data string in the send module and transmitting it to the read module/host computer for display on the monitor. This was accomplished by executing the read program on its host computer and taking the assigned or returned port number for use in executing the send program. The C function sscanf was added to the read program to facilitate the conversion of character strings to floats. An operator at the send terminal enters some numerical data which is read via keyboard as floating point numbers. A multiplication operation is performed on the data and the results are converted to strings before it

is transmitted to the read or receiving terminal. As in the previous test, the read program has already been executed and has been waiting for data. The port number has been fixed in the read module, this time, otherwise it would change for each execution. When the read module receives the data from the sending computer its converts the data from a string back to floating point numbers and displays them on the monitor. This demonstration shows how sockets could be used to transmit processed and unprocessed data among robots. The merge of the socket read program with the application program is provided as Appendix E. Figure 7 is the corresponding functional flow diagram. This shell provides a "plug and play" feature for integration with various task programs. Expansion of the above test for a single robot to multiple robots requires the modification of the socket_send module for addressing more than one robot. The robot simulator was required for testing and is discussed in Chapter VII.

B. SLEEP EVALUATION

Appendix A is the original SLEEP program, sleep.c, as provided by Gage. It was written in Symantec Think C on an Apple Macintosh computer. The test plan for SLEEP evaluation consists of determining the state of the code by making it portable for execution on the resident UNIX network, determining the syntax that it recognizes, providing LISP-like arguments for it to evaluate, and integrating its rule

base and expression buffers into the robot application program. An overall evaluation can then be conducted by transmitting SLEEP commands through the socket_send module in a simulated and real environment.

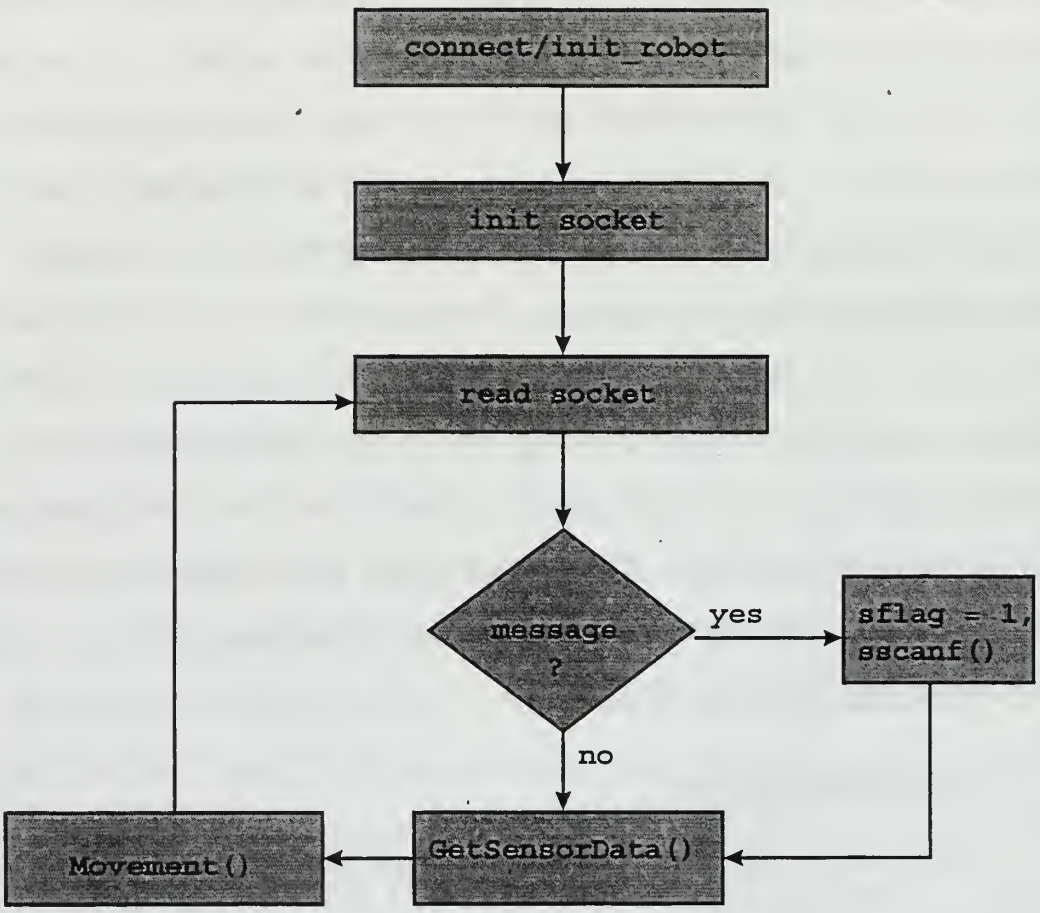

Figure 7. Socket/Application Functional Flow Diagram

VI. RESULTS

A. SLEEP EVALUATION

1. Compilation and Portability

The initial attempt to compile sleep.c was unsuccessful because of a portability deficiency. Portability refers to the ability to write a program on one computer platform and run it on another computer platform with little or no program modifications [Ref. 22]. The original code uses the console header file, console.h which is not a part of the local network /include directory. SUN OS release 4.1.3_U1 is the UNIX version that was used throughout the study. Importing console.h into a local directory did not enable compiling since the local network does not contain the csetmode function which is used by console.h. Since console.h appears to just encapsulate one of the standard header files, stdio.h, it was discarded and the SLEEP program was compiled by the GNU gcc compiler. The cc compiler can not be used since it does not contain stdarg.h which is used in sleep.c.

2. Syntax Determination

This level of testing was comparable to travelling abroad and attempting to learn the native or host language. Without fluency, the guest may attempt to pronounce words of the language and form meaningful sentences. Feedback or gestures from the host helps the guest determine a comprehensible syntax. Similiarly, expressions were entered

for SLEEP until something was returned. The goal was not the determination of useful SLEEP expressions but just to get the program to accept an input or expression and acknowledge its reception by echoing it back and parsing it. This is analogous to the host saying "This is what I heard you say: blah, blah, blah" even though the phrases make no sense in the context of the native language.

Sleep.c was modified to accomplish the above scenario. For dialogue to occur the listener must be able to distinguish between words and sentences. The listener must know when the speaker has finished a sentence. Accordingly, SLEEP must be able to distinguish between expressions and the end of a series of expressions. The character, @, denotes the end of an expression whereas the character, $, denotes the end of a series of expressions. These modifications are realized in sleepy.c, Appendix F . Without these, expression evaluation will not occur since the sleep.c will be waiting for character entry via the keyboard. This is akin to the above analogy where the host is still listening and not talking for he/she thinks the guest has more to say.

3. Expression Evaluation

This part of the test plan examined the interpreting ability of SLEEP. Given a LISP-like expression for evaluation, what result would SLEEP return? The test plan consists of experimentation with some of the operators that

are defined in the function string, *funStr[], of the SLEEP program, sleepy.c. Some test examples and results are as follow:

Input:(- 5 6 9)

Output: eval = -10

Input:(* (/ 1 4 2) (* 2 6) (+ 3 7))

Output: eval = 168

Input:(SETQ a (5))

Output: ss = (SETQ BAD (5)), eval 32000, L* eval(SETQ) found 32010 as dest, eval=0.

(The "BAD" and "eval=0" are returned since "a" is not defined in the variable string, *varStr[].)

Input:(SETQ E2 40000)

Output: ss = (SETQ E2 (-25536)), eval 32000, L* eval(SETQ) found 31002 as dest, eval=0.

(40000 is out of the range of of expStr[] so it turned into a negative number.)

Input:(SETQ E2 (30500))

Output: segmentation fault.

(30500 is greater than VARBASE so the operation is illegal.)

Input:(OR 0 2 0)

Output: eval=2

Input:(OR 3 0 5)

Output: eval=3

(OR evaluates its arguments from left to right. The first non-NIL value is returned. Any remaining arguments will not be evaluated. Otherwise OR returns NIL [Ref. 23].)

Input:(AND 1 3 0 5)

Output: eval=0

Input:(AND 1 3 5)

Output: eval=5

(AND evaluates its arguments from left to right. If a NIL condition is encountered, then it is returned and the remaining arguments won't be evaluated. Otherwise AND returns the value of its last argument [Ref. 23].)

The LISP operator COND has the general argument form (<test1>...<result1>) ... (<test n> ... <result n>) where each test is evaluated for a non-NIL value. The last result for the successful test will be returned as the overall result for COND [Ref. 23].

Input:(COND(99))

Output: eval=99

(as it should; this is good!)

Input: (COND(V2)(E2))

Output: eval=32

(It returned the value of the first true argument.)

Input: (COND(V1)(V2))

Output: eval=31

(It returned the value of the first true argument.)

Input: (COND ((AND 1 1) 99) ((AND 1 0) 88))

```
Output: eval=99
(It returned 99 since its corresponding test was a success.)
Input: (COND ((AND 1 0) 99) ((AND 1 1) 88))
Output:  ss=(COND ((AND 1 0) 99) ((AND 1 1) 88)),ss=(COND
((AND 1 0) 99) ((AND 1 1) 88)), eval 32000, eval 32000, eval
1, eval 0, L* eval(COND) found COND instead of '(', eval=0
```

It has not been determined why the last test failed. In accordance with LISP convention , the second test is successful or true so the second result, 88, should be returned. Sleepy.c was modified to provide results at each stage of the evaluation instead of returning only the overall expression evaluation. Currently, the SLEEP program is not asynchronous or non-blocking. It can't perform successive evaluations. The program must be re-initialized for each evaluation. This does not allow complete testing of the SETQ operation since "memory" is required to read back the variable to confirm the value that it was assigned or set to. Although, the SLEEP code is still in an exploratory form, its potential functionality has been demonstrated and it deserves attention for further development. Because of its present form and time constraints, the SLEEP code was not integrated with the socket-robot program but other approaches were taken to gauge the SLEEP concept feasibility.

B. BROADCAST MODULE

The sockets communication channel can be configured for broadcast support by using the setsockopt()routine; however, this option was not pursued since it would interfere with the Department network. Instead, the socket_send module was altered for the simultaneous creation and use of two sockets. The program, brdcst.c is provided in Appendix G.

C. INTEGRATED PROGRAM TESTING

Under the SLEEP concept, an expression is broadcast and all agents would evaluate the expression and execute it if it applies. This requires the SLEEP rule base onboard all agents. A similiar test without the SLEEP rule base was conducted using two NOMAD SCOUT mobile robots. The application programs rob1.c and rob2.c of Appendices H and I, respectively, are identical with the exception of TCP/IP port assignments for correspondence to the mobile robots named SCOUT1 and SCOUT2. The application program contains an algorithm for stopping when within ten inches of an obstacle and when in receipt of a message, any message, from the socket_send module.

Although it was not done in this test, the message sent could be the desired docking or stand-off range. Similiar test for changing the velocities were successfully completed. The outcome of this test depends upon how each robot "perceives" or senses its environment. As with the SLEEP concept, it is expected that only those robots that

meet the criteria will respond or react and deviate from their routine evolution. This test consisted of two phase: simulation then real world.

1. Simulation

The Nomadics robot simulator was used for the creation of a room with barriers. The robots' initial starting positions are as shown in Figure 8. The remote sending station was realized via rlogin from the local/host station. After simultaneous programs execution, the robots were allowed to roam and demonstrate collision avoidance. Once both robots were in open area, a message was sent. Confirmation of message reception can be observed in the local station's command window. "I got a message" is displayed upon message receipt. Otherwise, "no message this time" scrolls up in the command window. Range to the closest obstacle and the corresponding sensor # is also continuously given in the local machine's command window.

Figure 9 shows the final stopping positions of both robots. SCOUT2 is in the upper left corner was the first robot to stop. Both robots stopped at a distance of 9 inches. In the world map(previous figure) one grid length is 24 inches. These are favorable results since the one inch difference between the actual distance and the ten inch threshold is probably a function or result of robot velocity and system delay or response. Successive simulations also provide favorable results.

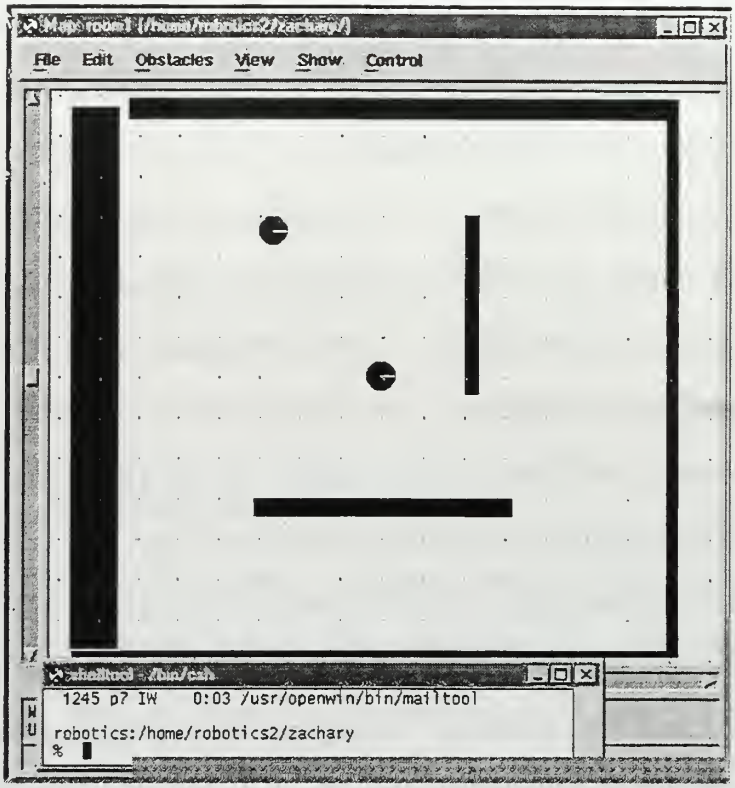

Figure 8. Initial State of Simulation

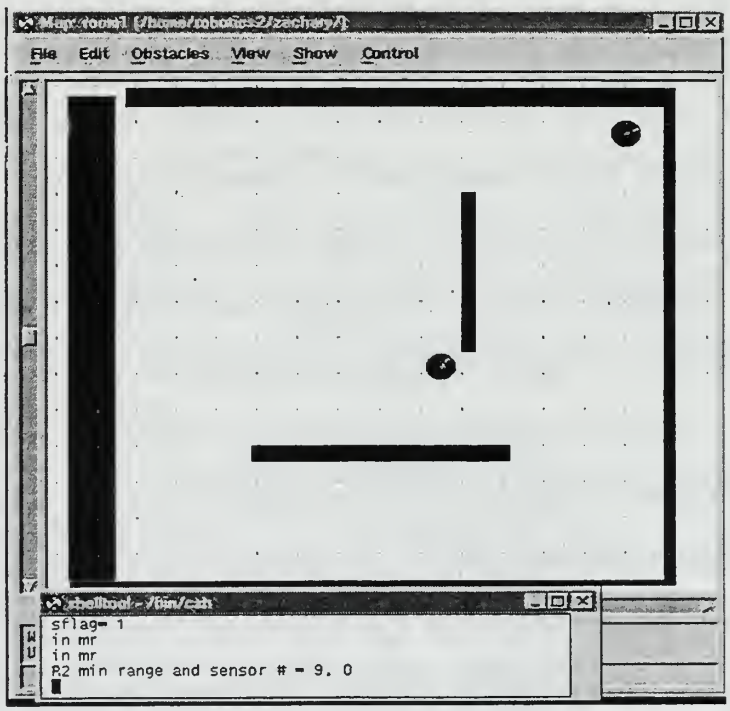

Figure 9. Final State of Simulation

2. Real World

This test was conducted in the Naval Postgraduate School Electrical Engineering Servo & Controls/Robotics Laboratory as shown in Figure 10. The procedure used in the simulated test was also used here. Unlike the simulated case, either one or both robots would stop shortly after receiving the message. The ranges to the closest objects, as shown in the command window, were less than ten inches but the actual distance was greater than ten inches. A sonar sensors calibration was conducted to determine if the problem was software or hardware related. Each robot was placed in the lab aisle or corridor, its sonars were continuously fired, and the ranges were observed from the command window. This was done while the robot was stationary and also while the robot was remotely rotated via joystick. Comparison of the observed ranges to actual ranges showed that a few ranges that were as small as five inches were in error. Theoretically, this could be the cause of the problem but what was its source? By decreasing the firing rate of the sonars, the range accuracy consistency improved. The sonar firing rate can be set between 0 and 1.02 seconds in 4 millisecond intervals. This can be adjusted in the application program by setting the "firerate" argument of the sensor parameters setting command, conf_sn [Ref. 24]. Firerate or the period between firings, starts after the end of the previous sonar return

processing which depends on time of flight. After changing firerate from 1 to 6, better test results were obtained. The opposite of multi-path may have occurred. At smaller intervals between firings, one sonar echo may arrive in the processing window of a second sonar right after the departure of the second sonar's incident wave. The second sonar thinks this echo is its own and calculates a range shorter than the actual range. Two trials were conducted. In trial 1, one robot stopped at five inches while the other one stopped at seven inches. In trial 2, one robot stopped at seven while the other stopped at twelve inches. Increasing the time between sonar firings reduces the interference but provides a larger delay between range updates and thereby reduces the system response. This occurrence, along with inherent system processing delay and velocity magnitudes, will affect overall performance, precision, and accuracy.

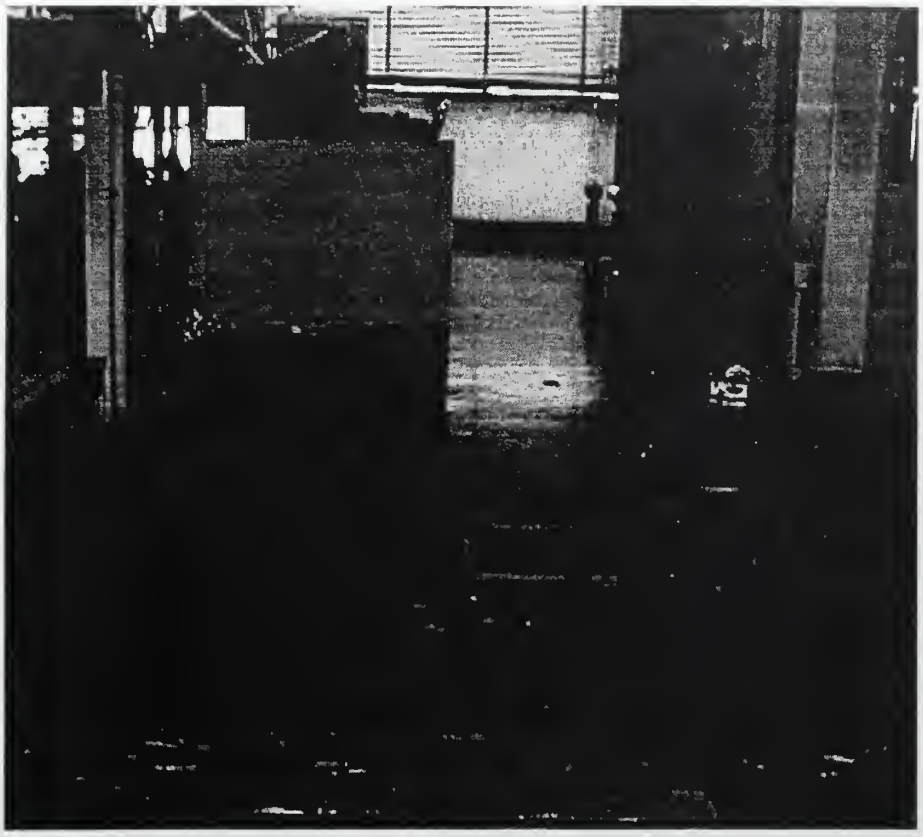

Figure 10. Real-world Test Environment

VII. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE STUDY

This thesis investigated some issues involved in the development of a multi-robot command and control architecture. Specific accomplishments are as follow:

1. Created a socket Interprocess communications (IPC) facility.
2. Integrated the socket IPC with a robot application program to develop a testbed system.
3. Ported the SLEEP code to be compatible with the campus LAN and conducted a static evaluation of it.
4. Tested and evaluated a SLEEP-like communications scheme in simulations and real-world test with two NOMAD SCOUT robots.

A level of coordination for cooperative group behavior can be achieved by implementing an appropriate communications scheme. Sockets were used to provide an IPC facility for dynamic addressing and grouping among multiple robots. This communications scheme led to favorable test results in support of achieving some aspects of group behavior. A static evaluation of SLEEP along with SLEEP-like simulations and real-world test demonstrated its capability as a viable diagnostic and control tool. Portability must be considered in the course of system development for the generations of developers and systems that will be needed.

Future studies may consider to extend and integrate the SLEEP code for dynamic testing on multiple robots as a diagnostic tool and for inter-robot communications. Appendix J provides some code segments for linking variables between the application program and the SLEEP module [Ref. 10]. A workcell for task development is needed to focus developmental efforts towards practical applications. The convergence of a swarm of robots upon one point or through an opening may be studied along with its battlefield applications such as overwhelming any opposing forces [Ref. 10]. A step towards this could begin with the simulation and real-world test of multiple robots leaving a room. This will surely provide additional insight to the relation of communications and cooperation.

A network or subnet may be created for testing the command broadcast. Any inherent difference between using multiple sockets and a broadcast may certainly become apparent during testing. The Linux operating system may be incorporated on a notebook PC or PDA for the development of mobile, stand-alone operations. The era of network-centric warfare warrants the analysis of how autonomous agents should link or fit into IT-21 and the Global Command and Control System (GCCS). Military forces can not fully "do more with less" without continuous addressal of these issues.

APPENDIX A. SLEEP.C

```
/* SLEEP.C: This is the original, exploratory version of SLEEP as
conceived by Dr. Douglas W. Gage, SPAWARSYSCEN San Diego, CA */

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <console.h>
#include <math.h>
#include <string.h>

#define NUMVAR 6
#define NUMEXP 6
#define NUMFUN 13
#define VARBASE 30000
#define EXPBASE 31000
#define FUNBASE 32000
#define DONE 32100

#define F_LFPAREN 0
#define F_RTPAREN 1
#define F_SETQ 2
#define F_COND 3
#define F_PLUS 4
#define F_MINUS 5
#define F_MULT 6
#define F_DIV 7
#define F_AND 8
#define F_OR 9
#define F_BAD 10
#define F_SETQQ 11
#define F_PRINT 12

char *funStr[] = { "(",")","SETQ", "COND","+","-","*","/","AND",
        "OR","BAD","SETQQ","PRINT" };

short var[20] = { 30, 31, 32, 33, 34, 35, 36, 37 };

char *varStr[] = { "V0", "V1", "V2", "V3", "V4", "V5" };
char *expStr[] = { "E0", "E1", "E2", "E3", "E4", "E5" };

typedef short expr[225];

expr ex[20];

void Log (char *fmt, ...)
{
char s[255];
va_list args;
va_start (args, fmt);
vsprintf (s,fmt,args);
printf ("\nL* %s\n", s);
va_end (args);
}
```

```
void printItem (short *t);

short *final (short *t) {
// if *t is '(', returns ptr to balancing ')'; otherwise returns t
short depth = 0;

if (*t == FUNBASE + F_LFPAREN) depth++;
if (*t == FUNBASE + F_RTPAREN) depth--;
if (depth < 0)  Log ("skip() encountered erroneous ')'");
while (depth > 0)  {
        t++;
        if (*t == FUNBASE + F_LFPAREN) depth++;
        if (*t == FUNBASE + F_RTPAREN) depth--;
        }
return t;
}

short eval (short **t) {
// performs DEEP evaluation, EVALing all expressions named, to "any"
depth (?)
// detects cycles, and calcs EVAL of any expression or variable only
once
// evaluates left to right, depth first

static short depth;
short tt, funCode, numArgs, accum, dest, done, i;
short *tp, *ep, *ep1, **ep2;

depth++;
tt = **t;
printf("eval %d\n", tt);
if (tt < VARBASE)  accum = tt;
// number
else if (tt < EXPBASE) accum = var[tt - VARBASE];
// variable
else if (tt < FUNBASE)  {
// expression
        ep2 = &ep1;
        ep1 = ex[tt - EXPBASE];
        accum = eval(ep2);
        }
else  {
// function
        funCode = tt - FUNBASE;
        if (funCode != F_LFPAREN)
                { Log ("eval() found %s without '('", funStr[funCode]);
return 0;  }
        (*t)++;
        tt = **t;
        if (tt < FUNBASE) { Log ("eval() found token %d after '('", tt);
return 0;  }
        funCode = tt - FUNBASE;
        numArgs = 0;
        done = 0;
        accum = 0;
```

```
        switch (funCode)  {
// special case setup
                        case F_MULT:
                        case F_DIV:
                        case F_AND:
                        accum = 1;
                        }
        (*t)++;
        tt = **t;
        while (tt != FUNBASE + F_RTPAREN)  {
// for each arg
                numArgs++;
                if (done) *t = final(*t);
                else switch (funCode)  {
                        case F_PLUS: accum = accum + eval(t); break;
                        case F_MINUS:
                                if (numArgs == 1) accum = eval(t);
                                else accum = accum - eval(t);
                                break;
                        case F_MULT: accum = accum * eval(t); break;
                        case F_DIV:
                                if (numArgs == 1) accum = eval(t);
                                else accum = accum / eval(t);
                                break;
                        case F_AND:
                                if ((accum = eval(t)) == 0) done = 1;
                                break;
                        case F_OR:
                                if (accum = eval(t)) done = 1;
                                break;
                        case F_COND:
                                if (**t != FUNBASE + F_LFPAREN)
                                        { Log ("eval(COND ) found %s
instead of '('", funStr[funCode]);  return 0; }
                                (*t)++;
                                if (accum = eval(t))  {
                                        (*t)++;
                                        while (**t != FUNBASE +
F_RTPAREN)  {
                                                accum = eval(t);
                                                (*t)++;
                                                }
                                        done = 1;
                                        }
                                else  {
                                        while (**t != FUNBASE +
F_RTPAREN)  {
                                                *t = final(*t);
                                                (*t)++;
                                                }
                                        }
                                break;
                        case F_SETQ:
                                if (((dest = **t) < VARBASE) || (dest >=
EXPBASE))
                                        {  Log ("eval(SETQ ) found %d as
```

```
dest", dest);  return 0;  }
                                (*t)++;
                                var[dest - VARBASE] = eval(t);
                                break;
                        case F_SETQQ:
                                if (((dest = **t) < EXPBASE) || (dest >=
FUNBASE))
                                        {  Log ("eval(SETQQ ) found %d
as
dest", dest);  return 0;  }
                                (*t)++;
                                tp = *t;
                                ep = ex[dest - EXPBASE];
                                *t = final(*t);
                                while (tp <= *t)  *ep++ = *tp++;
                                break;
                        case F_PRINT:
                                printItem(*t);
                                break;
                        }
                if (numArgs > 100) { Log ("eval numArgs > 100");  return
0;  }
                (*t)++;
                tt = **t;
                }
        switch (funCode)  {
// special case cleanup
                        case F_MINUS:
                                if (numArgs == 1) accum = - accum;
                                break;
                        case F_DIV:
                                if (numArgs == 1) accum = 1 / accum;
                                break;
                        case F_PRINT:
                                if (numArgs == 0)  {
                                        for (i = 0; i < NUMVAR; i++)  {
                                                *tp = VARBASE + i;
                                                printItem (tp);
                                                }
                                        for (i = 0; i < NUMEXP; i++)  {
                                                *tp = EXPBASE + i;
                                                printItem (tp);
                                                }
                                        }
                                break;
                        }
        }
depth--;
return accum;
}

void disp (char *s, short *t) {
char tempStr[20];
short oldT, *tt;

*s = 0;
```

```
oldT = F_LFPAREN;
tt = final(t);
while (t <= tt) {
        if ((oldT != (FUNBASE + F_LFPAREN)) && (*t != (FUNBASE +
F_RTPAREN)))
                strcat (s, " ");
// space
        if (*t < VARBASE)  {
// number
                sprintf (tempStr, "%d", *t);
                strcat (s, tempStr);
                }
        else if (*t < EXPBASE) strcat (s, varStr[*t - VARBASE]);
// variable
        else if (*t < FUNBASE) strcat (s, expStr[*t - EXPBASE]);
// expression
        else strcat (s, funStr[*t - FUNBASE]);
// function
        oldT = *t++;
        }
}

void printItem (short *t) {
char s[255];

if (*t < VARBASE) printf ("NUM %d\n", *t);
// number
else if (*t < EXPBASE)
        printf ("VAR %s = %d\n",varStr[*t - VARBASE], var[*t -
VARBASE]);
// variable
else if (*t < FUNBASE)  {
        disp (s, ex[*t - EXPBASE]);
        printf ("EXP %s = %s\n",expStr[*t - EXPBASE], s);       //
expression
        }
else printf ("FUN %s\n",funStr[*t - FUNBASE]);                  //
function
}

short parse (char *s, short *ep) {
char tok[25], *t;
short i, done = 0;

t = tok;
while (done == 0) {
        if (*s == 0) done = 1;
        if ((*s == 0) || (*s == ' ') || (*s == '(') || (*s == ')'))  {
                if (t > tok)  {
                        *t = 0;
                        if (((*tok >= '0') && (*tok <= '9')) || ((*tok
==
'-') && (t > (tok + 1))))
                                *ep++ = atoi (tok);
                        else  {
                                for (i = 0; i < NUMFUN; i++) if
```

```
(strcmp (tok, funStr[i]) == 0) break ;
                                if (i < NUMFUN) *ep++ = FUNBASE + i;
                                else  {
                                        for (i = 0; i < NUMVAR; i++) if
(strcmp (tok, varStr[i]) == 0) break;
                                        if (i < NUMVAR) *ep++ = VARBASE
+ i;
                                        else  {
                                                for (i = 0; i < NUMEXP;
i++) if (strcmp (tok, expStr[i]) == 0) break;
                                                if (i < NUMEXP) *ep++ =
EXPBASE + i;
                                                else *ep++ = FUNBASE +
F_BAD;
                                                }
                                        }
                                }
                        }
                if (*s == '(')  *ep++ = FUNBASE + F_LFPAREN;
                else if (*s == ')') *ep++ = FUNBASE + F_RTPAREN;
                t = tok;
                }
        else *t++ = toupper(*s);
        s++;
        }
*ep = DONE;
}

main () {
short *e1, **e2;
char ss[255], c;
short i;
short done = 0;
csetmode (C_RAW, stdin);

ex[1][0] = 125;
ex[1][1] = 32100;
while (done == 0)  {
i = 0;
while ((c = getchar()) != 13) if (c != -1)  {          // wait for char

        if (c == 0x7) done = 1;
        else if (c == 0x8) i--;
        else  ss[i++] = c;
        printf("%c", c);
        }
ss[i] = 0;
// <cr> falls thru ...
printf ("ss = %s\n", ss);

parse (ss, ex[0]);
disp (ss, ex[0]);
printf ("ss = %s\n", ss);

e2 = &e1;
```

```
e1 = ex[0];
printf (" eval = %d\n", eval(e2));

}

printf ("done");

}
```

APPENDIX B. TUNE-UP.C

```
/*************************************************************
 *
 * PROGRAM: Tune-up.c
 *
 * PURPOSE: Demonstrate basic client interface to Nserver, as well as
 * the reading of sensors and simple motion control. The robot does
 * nothing useful except for usually not hitting things.
 *
 *************************************************************/


/*** Include Files ***/

#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* macros to convert Nomad 200 motion commands to Scout motion commands
*/
#define ROTATION_CONSTANT 0.118597  /* inches/degree (Known to 100 ppm)
*/

#define RIGHT(trans, steer)
(trans+(int)((float)steer*ROTATION_CONSTANT))
#define LEFT(trans, steer)  (trans -
(int)((float)steer*ROTATION_CONSTANT))

#define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
steer), 0)
#define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
steer), 0)



/*** Constants ***/

#define TRUE 1
#define FALSE 0


/*** Function Prototypes ***/

void GetSensorData(void);
void Movement(void);


/*** Globals ***/

long SonarRange[16];  /* array of sonar readings (inches) */
long IRRange[16];  /* array of infrared readings (no units) */
int BumperHit = 0;  /* boolean value */
```

```
/*** Main Program ***/

main (unsigned int argc, char** argv)
{
  int i, index;
  int oldx, oldy;
  int order[16];


  SERV_TCP_PORT = 7771;

  /* Connect to Nserver. The parameter passed must always be 1. */
  connect_robot(1);


  /* Initialize Smask and send to robot. Smask is a large array that
  controls which data the robot returns back to the server, This
  function tells the robot to give us everything. */
  init_mask();


  /* Configure timeout (given in seconds). This is how long the robot
  will keep moving if you become disconnected. Set this low if there
  are walls nearby. */
  conf_tm(1);


  /* Sonar setup: configure the order in which individual sonar units
     fire. In this case, fire all units in counter-clockwise order
     (units are numbered counter-clockwise starting with the front
     sonar as zero). The conf_sn() function takes an integer and an
     array of at most 16 integers. If less than 16 units are to be
     used, the list must be terminated by a element of value -1. See
     the IR setup below for an example of this. The single integer
     value passed controls the time delay between units in multiples
     of four milliseconds. */
  for (i = 0; i < 16; i++)
    order[i] = i;
  conf_sn(1,order);

  /* Zero the robot. This aligns the turret and steering angles. The
     repositioning is neccessary to allow the user to position
     the robot where it was. */
  oldx = State[34];  /* remember position */
  oldy = State[35];
  zr();  /* tell robot to zero itself */
  ws(1,1,1,20);  /* wait until done zeroing */
  place_robot(oldx, oldy, 0, 0);  /* reposition simulated robot */


  /* Main loop. */
  while (!BumperHit)
    {
      GetSensorData();
      Movement();
    }
```

```
  /* Disconnect. */
  disconnect_robot(1);
}


/* Movement(). This function is responsible for using the sensor
   data to direct the robot's motion appropriately. */

void Movement (void)
{
  int i;
  int minreturn;
  int panic;
  int tvel, svel;


  /* Make sure we are not about to plow into something; check the
     front sonar and infrared sensors. If it looks bad, set panic
     flag. The threshold value for IRRangle has no exact physical
     relevance, and was empirically determined. */
  panic = FALSE;
  for (i = 12; i <= 15; i++)
    if (SonarRange[i] < 8 || IRRange[i] < 10) panic = TRUE;
  for (i = 0; i <= 4; i++)
    if (SonarRange[i] < 8 || IRRange[i] < 10) panic = TRUE;


  /* Move forward if we're not about to hit something. */
  if (!panic)
    tvel = 78;  /* can be between 0 and 280 */
  else
    tvel = 0;


  /* Determine the closest sonar return. Since the robot will only be
     moving forward, we only really need to worry about the front 8
     sensors. */
  minreturn = 12;
  for (i = 12; i <= 15; i++)
    if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;
  for (i = 0; i <= 4; i++)
    if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;


  /* Decide how to move.  There are three situations: 1) panic, 2) not
panic but
     near something, 3) clear to move. */

  if (panic)
    {
      tvel = 0;  /* if panic, stop moving and make a big turn. */
        svel = 340;
```

```
    }
  else if (SonarRange[minreturn] < 35) /* It is near something */
    {
      if (minreturn > 8)              /* obstacle on the right */
        {                             /* steer left    */
           svel = 240;
           tvel = 50;
        }
      else                            /* obstacle on the left */
        {                             /*  steer right        */
           svel = -240;
           tvel = 50;
        }
    }
   else  /* it is clear to move */
    {
       svel = 0;
       tvel = 78;
    }


  /* Set the robot's velocities. The first parameter is the robot's
     translational velocity, in tenths of an inch per second. This
     velocity can be between -240 and 240. The second parameter is the
     steering velocity, in tenths of a degree per second, and
     can be between -450 and 450. */
  scout_vm(tvel,svel);
}


/* GetSensorData(). Read in sensor data and load into arrays. */
void GetSensorData (void)
{
  int i;


  /* Read all sensors and load data into State array. */
  gs();


  /* Read State array data and put readings into individual arrays. */
  for (i = 0; i < 16; i++)
    {
     /* Sonar ranges are given in inches, and can be between 6 and
     255, inclusive. */
     SonarRange[i] = State[17+i];

    }


  /* Check for bumper hit. If a bumper is activated, the corresponding
     bit in State[33] will be turned on. Since we don't care which
     bumper is hit, we thus only need to check if State[33] is greater
     than zero. */
  if (State[33] > 0)
    {
```

```
BumperHit = 1;
tk("Ouch.");
printf("Bumper hit!\n");
}
                                    }
```

APPENDIX C. SOCKET_READ.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>


/*
* In the included file <netinet/in.h> a sockaddr_in is defined as
* follows:
* structure sockaddr_in {
*     short sin_family;
*     u_short   sin_port;
*     struct in_addr sin_addr;
*     char sin_zero[8];
* };
*  This program creates a datagram socket, binds a name to it, then
*  reads from
*  the socket.
*/
main()
{
      int sock, Length;
      struct sockaddr_in name;
      char buf[1024];
        float km,kmy,kmt,x,y,theta;


      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
          perror ("opening datagram socket");
          exit(1);

      }
      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 4000;
      if (bind(sock, &name, sizeof(name))) {
         perror("binding datagram socket");
         exit(1);

      }
      /* Find assigned port value and print it out. */
      Length = sizeof(name);
      if (getsockname(sock, &name, &Length)) {
          perror("getting socket name");
          exit(1);

      }
      printf("socket has port #%d\n", ntohs(name.sin_port));
      /* Read from the socket */
```

```
        if (read(sock, buf, 1024) < 0)
              perror("receiving datagram packet");
        printf("-->%s\n", buf);

        sscanf(buf, "%f %f %f",&km, &kmy, &kmt);   /* convert string buf
to float km */


        close(sock);
}
```

APPENDIX D. SOCKET_SEND.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>


#define mi_to_km 1.2
#define km_to_m 1000

/* Send a datagram to a receiver whose name is obtained from the
 * command line arguments.
 * The form of the command line is socket_send hostname portnumber
 */


main(argc, argv)

      int argc;
      char *argv[];

{
      float x,y,theta,km,kmy,kmt;
      char d[128];
      int sock;
      struct sockaddr_in name;
      struct hostent *hp, *gethostbyname();

      printf("Enter and read distance in miles:\n");

      scanf("%f %f %f",&x, &y, &theta);


  /* miles-to-kilometer conversion; this is just a test/calculation
   example: */

        km=x*mi_to_km;
        kmy=y*mi_to_km;
        kmt=theta*mi_to_km;


      sprintf(d,"%12.2f %12.2f %12.2f",km,kmy,kmt);  /* convert number
    to string */


      /* Create socket on which to send. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
            perror("opening datagram socket");
          exit(1);
      }
      /*
       * Construct name, with no wildcards, of the sockets to send to.
       * Gethostbyname() returns a structure including the network
```

```
        * address of the specified host.  The port number is taken from
        * the cmd line
        */
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
              fprintf(stderr, "%s: unknown host0", argv[1]);
              exit(2);

        }
        bcopy(hp->h_addr, &name.sin_addr, hp->h_lengt );
          name.sin_family = AF_INET;
        name.sin_port = htons(atoi(argv[2]));

        /* Send message. */
        if (sendto(sock, d, sizeof(d), 0, &name, sizeof(name)) < 0)
            perror("sending datagram message");
        close(sock);
}
```

APPENDIX E. SOCKET_SHELL.C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <fcntl.h>



/*
* In the included file <netinet/in.h> a sockaddr_in is defined as
* follows:
* structure sockaddr_jn {
*      short sin_family;
*      u_short   sin_port;
*      struct in_addr sin_addr;
*      char sin_zero[8];
*      };
*  This program creates a datagram socket, binds a name to it, then
*  reads from the socket.
*
*/

main()
{
      int sock, Length;

      struct sockaddr_in name;
      char buf[1024];
        float km,kmy,kmt,x,y,theta;
        int flag=-1;

      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
          perror ("opening datagram socket");
          exit(1);
      }

        /* switch to asynch. mode  or non-blocking mode */
      fcntl(sock, F_SETFL, FNDELAY);


      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 4000;
      if (bind(sock, &name, sizeof(name))) {
         perror("binding datagram socket");
         exit(1);
      }
      /* Find assigned port value and print it out. */
```

```
        Length = sizeof(name);
        if (getsockname(sock, &name, &Length)) {
            perror("getting socket name");
            exit(1);

        }
          printf("socket has port #%d\n", ntohs(name.sin_port));
          /* printf("read1= %d\n", read(sock, buf, 1024)); */


        /* Read from the socket */
          while (1)
        {
        if (read(sock, buf, 1024) < 0)
             {
              /* do something without a message from other side */
              printf("No messag this time \n");
             }
        else
             {
               /* do something different with a message from other side */
                  printf("Yes, I got a message\n");

             }


               /* let robot perform a task here. */
        }
        printf("-->%s\n", buf);

        sscanf(buf, "%f %f %f",&km, &kmy, &kmt);
 /* convert string buf to    float km */

  close(sock);
  printf("end\n");
}
```

APPENDIX F. SLEEPY.C

```
/* Modified sleep.c code */


#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define NUMVAR 6
#define NUMEXP 6
#define NUMFUN 13
#define VARBASE 30000
#define EXPBASE 31000
#define FUNBASE 32000
#define DONE 32100

#define F_LFPAREN 0
#define F_RTPAREN 1
#define F_SETQ 2
#define F_COND 3
#define F_PLUS 4
#define F_MINUS 5
#define F_MULT 6
#define F_DIV 7
#define F_AND 8
#define F_OR 9
#define F_BAD 10
#define F_SETQQ 11
#define F_PRINT 12


char *funStr[] = { "(",")","SETQ", "COND","+","-","*","/","AND",
        "OR","BAD","SETQQ","PRINT" };

short var[20] = { 30, 31, 32, 33, 34, 35, 36, 37 };

char *varStr[] = { "V0", "V1", "V2", "V3", "V4", "V5" };
char *expStr[] = { "E0", "E1", "E2", "E3", "E4", "E5" };

typedef short expr[225];

expr ex[20];

void Log (char *fmt, ...)
{
char s[255];
va_list args;
va_start (args, fmt);
vsprintf (s,fmt,args);
printf ("\nL* %s\n", s);
va_end (args);
}
```

```
void printItem (short *t);

short *final (short *t) {
// if *t is '(', returns ptr to balancing ')'; otherwise returns t
short depth = 0;

if (*t == FUNBASE + F_LFPAREN) depth++;
if (*t == FUNBASE + F_RTPAREN) depth--;
if (depth < 0)  Log ("skip() encountered erroneous ')'");
while (depth > 0)  {
        t++;
        if (*t == FUNBASE + F_LFPAREN) depth++;
        if (*t == FUNBASE + F_RTPAREN) depth--;
        }
return t;
}

short eval (short **t) {
// performs DEEP evaluation, EVALing all expressions named, to "any"
depth (?)
// detects cycles, and calcs EVAL of any expression or variable only
once
// evaluates left to right, depth first

static short depth;
short tt, funCode, numArgs, accum, dest, done, i;
short *tp, *ep, *ep1, **ep2;

depth++;
tt = **t;
printf("eval %d\n", tt);
if (tt < VARBASE)  accum = tt;
// number
else if (tt < EXPBASE) accum = var[tt - VARBASE];
// variable
else if (tt < FUNBASE)  {
// expression
        ep2 = &ep1;
        ep1 = ex[tt - EXPBASE];
        accum = eval(ep2);
        }
else  {
// function
        funCode = tt - FUNBASE;
        if (funCode != F_LFPAREN)
                { Log ("eval() found %s without '('", funStr[funCode]);
return 0;  }
        (*t)++;
        tt = **t;

      printf("eval function = %d\n", tt); /* new line */
      if (tt < FUNBASE) { Log ("eval() found token %d after '('", tt);
return 0;  }
        funCode = tt - FUNBASE;
        numArgs = 0;
```

```
        done = 0;
        accum = 0;
        switch (funCode)  {
// special case setup
                        case F_MULT:
                        case F_DIV:
                        case F_AND:
                        accum = 1;
                        }
        (*t)++;
        tt = **t;
        while (tt != FUNBASE + F_RTPAREN)  {
// for each arg
                numArgs++;
                if (done) *t = final(*t);
                else switch (funCode)  {
                        case F_PLUS: accum = accum + eval(t); break;
                        case F_MINUS:
                                if (numArgs == 1) accum = eval(t);
                                else accum = accum - eval(t);
                                break;
                        case F_MULT: accum = accum * eval(t); break;
                        case F_DIV:
                                if (numArgs == 1) accum = eval(t);
                                else accum = accum / eval(t);
                                break;
                        case F_AND:
                                if ((accum = eval(t)) == 0) done = 1;
                                break;
                        case F_OR:
                                if (accum = eval(t)) done = 1;
                                break;
                        case F_COND:
                                if (**t != FUNBASE + F_LFPAREN)
                                        { Log ("eval(COND ) found %s
instead of '('", funStr[funCode]);  return 0;  }
                                (*t)++;
                                if (accum = eval(t))  {
                                        (*t)++;
                                        while (**t != FUNBASE +
F_RTPAREN)  {
                                                accum = eval(t);
                                                (*t)++;
                                                }
                                        done = 1;
                                        }
                                else  {
                                        while (**t != FUNBASE +
F_RTPAREN)  {
                                                *t = final(*t);
                                                (*t)++;
                                                }
                                        }
                                break;
                        case F_SETQ:
                                if (((dest = **t) < VARBASE) || (dest >=
```

```
EXPBASE))
                                        {  Log ("eval(SETQ ) found %d as
dest", dest);  return 0;  }
                                (*t)++;
                                accum = var[dest - VARBASE] = eval(t);
                                /* new line */

                                break;
                        case F_SETQQ:
                                if (((dest = **t) < EXPBASE) || (dest >=
FUNBASE))
                                        {  Log ("eval(SETQQ ) found %d
as
dest", dest);  return 0;  }
                                (*t)++;
                                tp = *t;
                                ep = ex[dest - EXPBASE];
                                *t = final(*t);
                                while (tp <= *t)  *ep++ = *tp++;
                                break;
                        case F_PRINT:
                                printItem(*t);
                                break;
                        }
                if (numArgs > 100) { Log ("eval numArgs > 100");  return
0;  }
                (*t)++;
                tt = **t;
                }
        switch (funCode)  {
// special case cleanup
                        case F_MINUS:
                                if (numArgs == 1) accum = - accum;
                                break;
                        case F_DIV:
                                if (numArgs == 1) accum = 1 / accum;
                                break;
                        case F_PRINT:
                                if (numArgs == 0)  {
                                        for (i = 0; i < NUMVAR; i++)  {
                                                *tp = VARBASE + i;
                                                printItem (tp);
                                                }
                                        for (i = 0; i < NUMEXP; i++)  {
                                                *tp = EXPBASE + i;
                                                printItem (tp);
                                                }
                                        }
                                break;
                        }
        }
depth--;
printf("eval = %d\n", accum); /* new line */
return accum;
}
```

```
void disp (char *s, short *t) {
char tempStr[20];
short oldT, *tt;

*s = 0;
oldT = F_LFPAREN;
tt = final(t);
while (t <= tt) {
        if ((oldT != (FUNBASE + F_LFPAREN)) && (*t != (FUNBASE +
F_RTPAREN)))
                strcat (s, " ");
// space
        if (*t < VARBASE)  {
// number
                sprintf (tempStr, "%d", *t);
                strcat (s, tempStr);
                }
        else if (*t < EXPBASE) strcat (s, varStr[*t - VARBASE]);
// variable
        else if (*t < FUNBASE) strcat (s, expStr[*t - EXPBASE]);
// expression
        else strcat (s, funStr[*t - FUNBASE]);
// function
        oldT = *t++;
        }
}

void printItem (short *t) {
char s[255];

if (*t < VARBASE) printf ("NUM %d\n", *t);
// number
else if (*t < EXPBASE)
        printf ("VAR %s = %d\n",varStr[*t - VARBASE], var[*t -
VARBASE]);
// variable
else if (*t < FUNBASE)  {
        disp (s, ex[*t - EXPBASE]);
        printf ("EXP %s = %s\n",expStr[*t - EXPBASE], s);
//expression
        }
else printf ("FUN %s\n",funStr[*t - FUNBASE]);
//function
}

short parse (char *s, short *ep) {
char tok[25], *t;
short i, done = 0;

t = tok;
while (done == 0) {
        if (*s == 0) done = 1;
        if ((*s == 0) || (*s == ' ') || (*s == '(') || (*s == ')'))  {
                if (t > tok)  {
                        *t = 0;
                        if (((*tok >= '0') && (*tok <= '9')) || ((*tok
```

```
  == '-') && (t > (tok + 1))))
                                *ep++ = atoi (tok);
                        else  {
                                for (i = 0; i < NUMFUN; i++) if
(strcmp (tok, funStr[i]) == 0) break ;
                                if (i < NUMFUN) *ep++ = FUNBASE + i;
                                else  {
                                  for (i = 0; i < NUMVAR; i++) if
(strcmp (tok, varStr[i]) == 0) break;
                                  if (i < NUMVAR) *ep++ = VARBASE
+ i;
                                  else  {
                                    for (i = 0; i < NUMEXP;
i++) if (strcmp (tok, expStr[i]) == 0) break;
                                    if (i < NUMEXP) *ep++ =
EXPBASE + i;
                                    else *ep++ = FUNBASE +
F_BAD;
                                    }
                                  }
                                }
                        }
                if (*s == '(')  *ep++ = FUNBASE + F_LFPAREN;
                else if (*s == ')') *ep++ = FUNBASE + F_RTPAREN;
                t = tok;
                }
        else *t++ = toupper(*s);
        s++;
        }
*ep = DONE;
}

main () {
short *e1, **e2;
char ss[255], c;
short i, a;
short done = 0;
int aa;

// csetmode (C_RAW, stdin);

ex[1][0] = 125;
ex[1][1] = 32100;
while (done == 0)  {
i = 0;
while ((c = getchar()) != '@') if (c != -1)  {          // wait for char

        if (c == '$') done = 1;
        else if (c == 0x8) i--;
        else ss[i++] = c;
        printf("%c", c);
        }
  printf("\n I got a statement! \n");
ss[i] = 0;
// <cr> falls thru ...
printf ("ss = %s\n", ss);
```

```
parse (ss, ex[0]);


disp (ss, ex[0]);
printf ("ss = %s\n", ss);

e2 = &e1;
e1 = ex[0];
printf (" eval = %d\n", eval(e2));

}

printf ("done");

}
```

APPENDIX G. BRDCST.C

```
/* This program creates two sockets for simultaneous connection to two
 *robots */


#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>


#define mi_to_km 1.2
#define km_to_m 1000



main(argc, argv)

        int argc;
      /* int stat; */
      char *argv[];/* *opt; */


{
      float x,y,theta,km,kmy,kmt;
      char d[128], *opt;
      int sock1, sock2, stat;
      struct sockaddr_in name1, name2;
      struct hostent *hp1, *hp2, *gethostbyname();

      printf("Enter and read distance in miles:\n");

      scanf("%f %f %f",&x, &y, &theta);

     km=x*mi_to_km;
        kmy=y*mi_to_km;
        kmt=theta*mi_to_km;

      /* meters=km*km_to_m; */
      sprintf(d,"%12.2f %12.2f %12.2f",km,kmy,kmt);  /* convert number
to string */


        printf("what is d: %s \n",d);



      /* Create socket on which to send. */
      sock1 = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock1 < 0) {
            perror("opening datagram socket1");
```

```
            exit(1);
        }
        sock2 = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock2 < 0) {
              perror("opening datagram socket2");
            exit(1);
        }


        /*
         * Construct name, with no wildcards, of the sockets to send to.
         * Gethostbyname() returns a structure including the network
address
         * of the specified host.  The port number is taken from the cmd
line
        */
        hp1 = gethostbyname(argv[1]);
        if (hp1 == 0) {
              fprintf(stderr, "%s: unknown host0", argv[1]);
              exit(2);

        }
        hp2 = gethostbyname(argv[3]);
        if (hp2 == 0) {
              fprintf(stderr, "%s: unknown host0", argv[3]);
              exit(2);
        }


        bcopy(hp1->h_addr, &name1.sin_addr, hp1->h_length);
          name1.sin_family = AF_INET;
        name1.sin_port = htons(atoi(argv[2]));

        bcopy(hp2->h_addr, &name2.sin_addr, hp2->h_length);
          name2.sin_family = AF_INET;
        name2.sin_port = htons(atoi(argv[4]));



        /* Send messages. */
        if (sendto(sock1, d, sizeof(d), 0, &name1, sizeof(name1)) < 0)
            perror("sending datagram message1");
        close(sock1);

        if (sendto(sock2, d, sizeof(d), 0, &name2, sizeof(name2)) < 0)
            perror("sending datagram message2");
        close(sock2);

printf("end of Prog\n");

}

/* on sending machine type: name of this program hostmachine1 name port1
 *  number hostmachine 2 name port2 number */
```

APPENDIX H. ROB1.C

```
/*  This is the integrated application program for SCOUT1 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <fcntl.h>

#include "Nclient.h"
#include <math.h>

/* macros to convert Nomad 200 motion commands to Scout motion commands
*/
#define ROTATION_CONSTANT 0.118597  /* inches/degree (Known to 100 ppm)
*/

#define RIGHT(trans, steer)
(trans+(int)((float)steer*ROTATION_CONSTANT))
#define LEFT(trans, steer)  (trans -
(int)((float)steer*ROTATION_CONSTANT))

#define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
steer), 0)
#define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
steer), 0)



/*** Constants ***/

#define TRUE 1
#define FALSE 0


/*** Function Prototypes ***/

void GetSensorData(void);
void Movement(void);


/*** Globals ***/

long SonarRange[16];  /* array of sonar readings (inches) */
long IRRange[16];  /* array of infrared readings (no units) */
int BumperHit = 0;  /* boolean value */
int sflag; /* parameter change from socket */
float km,kmy,kmt,x,y,theta;


/*
```

```
* In the included filw <netinet/in.h> a sockaddr_in is defined as
follows:
* structure sockaddr_in {
*     short sin_family;
*     u_short   sin_port;
*     struct in_addr sin_addr;
*     char sin_zero[8];
* };
*  This program creates a datagram socket, binds a name to it, then
reads from
*  the socket.
*/
main()
{ int i, index;
  int oldx, oldy;
  int order[16];
  int sock, Length;

  struct sockaddr_in name;
  char buf[1024];



  SERV_TCP_PORT = 7771;

  /* Connect to Nserver. The parameter passed must always be 1. */
  connect_robot(1);


  /* Initialize Smask and send to robot. Smask is a large array that
  controls which data the robot returns back to the server. This
  function tells the robot to give us everything. */
  init_mask();


  /* Configure timeout (given in seconds). This is how long the robot
  will keep moving if you become disconnected. Set this low if there
  are walls nearby. */
  conf_tm(1);

    /* Sonar setup: configure the order in which individual sonar units
     fire. In this case, fire all units in counter-clockwise order
     (units are numbered counter-clockwise starting with the front
     sonar as zero). The conf_sn() function takes an integer and an
     array of at most 16 integers. If less than 16 units are to be
     used, the list must be terminated by a element of value -1. See
     the IR setup below for an example of this. The single integer
     value passed controls the time delay between units in multiples
     of four milliseconds. */
  for (i = 0; i < 16; i++)
    order[i] = i;
  conf_sn(6,order);
```

```
      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
          perror ("opening datagram socket");
          exit(1);
      }


        /* switch to asynch. mode */
      fcntl(sock, F_SETFL, FNDELAY);



      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 5001;
      if (bind(sock, &name, sizeof(name))) {
         perror("binding datagram socket");
         exit(1);



      }
      /* Find assigned port value and print it out. */
      Length = sizeof(name);
      if (getsockname(sock, &name, &Length)) {
          perror("getting socket name");
          exit(1);

      }
        printf("socket has port #%d\n", ntohs(name.sin_port));
        /* printf("read1= %d\n", read(sock, buf, 1024)); */


                    /* let robot do something here: main prog */

  /* Main loop. */
  while (!BumperHit)
    {
      if (read(sock, buf, 1024) < 0)
           {
            /* do something without a message from other side */
            printf("No messag this time \n");
           }
      else
           {
             /* do something different with a message from other side */
               printf("Yes, I got a message\n");

                   sflag=1;
      sscanf(buf, "%f %f %f",&km, &kmy, &kmt);   /* convert string buf
to float km */
           }
```

```
 printf("read= %d\n", read(sock,buf,1024));


      GetSensorData();
      Movement();
    }  /* end while (!bumperhit) */


  /* Disconnect. */
  disconnect_robot(1);

      printf("-->%s\n", buf);

      printf("we make it past rd line b4 msg snt\n");

           close(sock);
  printf("END OF MAIN\n");
  }


/* Movement(). This function is responsible for using the sensor
   data to direct the robot's motion appropriately. */

void Movement (void)
{
  int i;
  int minreturn;
  int panic;
  int tvel, svel;


  /* Make sure we are not about to plow into something; check the
     front sonar and infrared sensors. If it looks bad, set panic
     flag. The threshold value for IRRangle has no exact physical
     relevance, and was empirically determined. */
  panic = FALSE;
  for (i = 12; i <= 15; i++)
    if (SonarRange[i] < 8) panic = TRUE;
  for (i = 0; i <= 4; i++)
    if (SonarRange[i] < 8) panic = TRUE;

    printf("sflag= %d\n", sflag);



  /* Determine the closest sonar return. Since the robot will only be
     moving forward, we only really need to worry about the front 8
     sensors. */
  minreturn = 12;

  for (i = 12; i <= 15; i++)
    if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;

printf("in mr\n");
```

```
  for (i = 0; i <= 4; i++)
    if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;

printf("in mr \n");
printf("R1 min range and sensor # = %d, %d\n", SonarRange[minreturn],
minreturn);

      while ((SonarRange[minreturn] < 10) && sflag==1)
         {
       st();
         }
printf("after st\n");


  /* Decide which way (if any) to turn. 3 cases: panic; not panic but
near; clear */
  if (panic)  /* we're about to hit something */
    {
    tvel = 0;
    svel = 340;  /* steer hard left to get turned around */
    }
  else if (SonarRange[minreturn] < 15)  /* we're near something */
    {
      if (minreturn > 8)  /* object on right side of robot */
        {
      svel = 320;  /* steer left */
        tvel = 15;
        }
      else  /* on left */
        {
      svel = -320;  /* steer right */
        tvel = 15;
        }
    }
  else  /* we're clear */
    {
    svel = 0;
    tvel = 75;
     }


  /* Set the robot's velocities. The first parameter is the robot's
     translational velocity, in tenths of an inch per second. This
     velocity can be between -240 and 240. The second parameter is the
     steering velocity, and the third is the turret velocity. The
     units of the latter two are tenths of a degree per second, and
     can be between -450 and 450. The same value is given for these
     two so that the turret is always facing the direction of
     motion. */

  /* if (sflag==1)
  scout_vm(km,kmy);
else */
  scout_vm(tvel,svel);
```

```
printf("past vm\n");
}
/* GetSensorData(). Read in sensor data and load into arrays. */
void GetSensorData (void)
{
  int i;


  /* Read all sensors and load data into State array. */
  gs();


  /* Read State array data and put readings into individual arrays. */
  for (i = 0; i < 16; i++)
    {
      /* Sonar ranges are given in inches, and can be between 6 and
      255, inclusive. */
      SonarRange[i] = State[17+i];

    }



  /* Check for bumper hit. If a bumper is activated, the corresponding
     bit in State[33] will be turned on. Since we don't care which
     bumper is hit, we thus only need to check if State[33] is greater
     than zero. */
  if (State[33] > 0)
    {
      BumperHit = 1;
      tk("Ouch.");
      printf("Bumper hit!\n");
    }
} /* end getsensordata */
```

APPENDIX I. ROB2.C

```
/* This is the integrated application program for SCOUT2 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <fcntl.h>

#include "Nclient.h"
#include <math.h>

/* macros to convert Nomad 200 motion commands to Scout motion commands
*/
#define ROTATION_CONSTANT 0.118597  /* inches/degree (Known to 100 ppm)
*/

#define RIGHT(trans, steer)
(trans+(int)((float)steer*ROTATION_CONSTANT))
#define LEFT(trans, steer)  (trans -
(int)((float)steer*ROTATION_CONSTANT))

#define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
steer), 0)
#define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
steer), 0)



/*** Constants ***/

#define TRUE 1
#define FALSE 0


/*** Function Prototypes ***/

void GetSensorData(void);
void Movement(void);


/*** Globals ***/

long SonarRange[16];  /* array of sonar readings (inches) */
long IRRange[16];  /* array of infrared readings (no units) */
int BumperHit = 0;  /* boolean value */
int sflag; /* parameter change from socket */
float km,kmy,kmt,x,y,theta;
```

```
/*
* In the included filw <netinet/in.h> a sockaddr_in is defined as
follows:
* structure sockaddr_in {
*     short sin_family;
*     u_short   sin_port;
*     struct in_addr sin_addr;
*     char sin_zero[8];
* };
*  This program creates a datagram socket, binds a name to it, then
reads from
*  the socket.
*/
main()
{ int i, index;
  int oldx, oldy;
  int order[16];
  int sock, Length;

  struct sockaddr_in name;
  char buf[1024];



  SERV_TCP_PORT = 7771;

  /* Connect to Nserver. The parameter passed must always be 1. */
  connect_robot(2);


  /* Initialize Smask and send to robot. Smask is a large array that
  controls which data the robot returns back to the server. This
  function tells the robot to give us everything. */
  init_mask();


  /* Configure timeout (given in seconds). This is how long the robot
  will keep moving if you become disconnected. Set this low if there
  are walls nearby. */
  conf_tm(1);

    /* Sonar setup: configure the order in which individual sonar units
     fire. In this case, fire all units in counter-clockwise order
     (units are numbered counter-clockwise starting with the front
     sonar as zero). The conf_sn() function takes an integer and an
     array of at most 16 integers. If less than 16 units are to be
     used, the list must be terminated by a element of value -1. See
     the IR setup below for an example of this. The single integer
     value passed controls the time delay between units in multiples
```

```
   of four milliseconds. */
for (i = 0; i < 16; i++)
  order[i] = i;
conf_sn(6,order);


    /* Create socket from which to read. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror ("opening datagram socket");
        exit(1);
    }

      /* switch to asynch. mode */
    fcntl(sock, F_SETFL, FNDELAY);


    /* Create name with wildcards. */
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = 5002;
    if (bind(sock, &name, sizeof(name))) {
       perror("binding datagram socket");
       exit(1);

    }
    /* Find assigned port value and print it out. */
    Length = sizeof(name);
    if (getsockname(sock, &name, &Length)) {
        perror("getting socket name");
        exit(1);

    }
      printf("socket has port #%d\n", ntohs(name.sin_port));
      /* printf("read1= %d\n", read(sock, buf, 1024)); */


                /* let robot do something here: main prog */


/* Main loop. */
while (!BumperHit)
  {
    if (read(sock, buf, 1024) < 0)
         {
          /* do something without a message from other side */
          printf("No messag this time \n");
         }
    else
         {
           /* do something different with a message from other side */
```

```
                printf("Yes, I got a message\n");
                    sflag=1;
      sscanf(buf, "%f %f %f",&km, &kmy, &kmt);   /* convert string buf
to float km */
           }

 printf("read= %d\n", read(sock,buf,1024));


      GetSensorData();
      Movement();
    }  /* end while (!bumperhit) */


  /* Disconnect. */
  disconnect_robot(2);

      printf("-->%s\n", buf);

      printf("we make it past rd li   b4 msg snt\n");

      close(sock);
  printf("END OF MAIN\n");
  }


/* Movement(). This function is responsible for using the sensor
   data to direct the robot's motion appropriately. */

void Movement (void)
{
  int i;
  int minreturn;
  int panic;
  int tvel, svel;


  /* Make sure we are not about to plow into something; check the
     front sonar and infrared sensors. If it looks bad, set panic
     flag. The threshold value for IRRangle has no exact physical
     relevance, and was empirically determined. */
  panic = FALSE;
  for (i = 12; i <= 15; i++)
    if (SonarRange[i] < 8) panic = TRUE;
  for (i = 0; i <= 4; i++)
    if (SonarRange[i] < 8) panic = TRUE;

    printf("sflag= %d\n", sflag);
```

```
  /* Determine the closest sonar return. Since the robot will only be
     moving forward, we only really need to worry about the front 8
     sensors. */
  minreturn = 12;

  for (i = 12; i <= 15; i++)
    if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;

printf("in mr\n");

  for (i = 0; i <= 4; i++)
    if (SonarRange[i] < SonarRange[minreturn])
      minreturn = i;

printf("in mr \n");
printf("R2 min range and sensor # = %d, %d\n", SonarRange[minreturn],
minreturn);

      while ((SonarRange[minreturn] < 10) && sflag==1)
         {
       st();
         }
printf("after st\n");


  /* Decide which way (if any) to turn. 3 cases: panic; not panic but
near; clear */
  if (panic)  /* we're about to hit something */
    {
    tvel = 0;
    svel = 340;  /* steer hard left to get turned around */
    }
  else if (SonarRange[minreturn] < 15)  /* we're near something */
    {
      if (minreturn > 8)  /* object on right side of robot */
        {
      svel = 320;  /* steer left */
        tvel = 15;
        }
      else  /* on left */
        {
      svel = -320;  /* steer right */
        tvel = 15;
        }
    }
  else  /* we're clear */
    {
    svel = 0;
```

```
    tvel = 75;
     }


  /* Set the robot's velocities. The first parameter is the robot's
     translational velocity, in tenths of an inch per second. This
     velocity can be between -240 and 240. The second parameter is the
     steering velocity, and the third is the turret velocity. The
     units of the latter two are tenths of a degree per second, and
     can be between -450 and 450. The same value is given for these
     two so that the turret is always facing the direction of
     motion. */

  /* if (sflag==1)
  scout_vm(km,kmy);
else */
  scout_vm(tvel,svel);
printf("past vm\n");
}
/* GetSensorData(). Read in sensor data and load into arrays. */
void GetSensorData (void)
{
  int i;


  /* Read all sensors and load data into State array. */
  gs();


  /* Read State array data and put readings into individual arrays. */
  for (i = 0; i < 16; i++)
    {
      /* Sonar ranges are given in inches, and can be between 6 and
      255, inclusive. */
      SonarRange[i] = State[17+i];

    }



  /* Check for bumper hit. If a bumper is activated, the corresponding
     bit in State[33] will be turned on. Since we don't care which
     bumper is hit, we thus only need to check if State[33] is greater
     than zero. */
  if (State[33] > 0)
    {
      BumperHit = 1;
      tk("Ouch.");
      printf("Bumper hit!\n");
    }
} /* end getsensordata */
```

APPENDIX J. CODE SEGMENTS FOR LINKING

This code may be used to link variables between the SLEEP program and the robot application program.

First, in the C code that implements the robot (reading sensors, controlling actuators, etc), declare ALL the variables that are to be referred in SLEEP with:

```
struct {
     short temp;    /* these names can be anything the user
     short x;            wants them to be  */
     short velocity;
} v;

short *var;
```

and declare varname[]. Let's say that they are "temp", "x", and "velocity" in that order.

when starting to execute, point var to v with:

```
var = &v;
```

Now, when using the string token "velocity" in a SLEEP expression, it will be recognized as varname[2], and SLEEP will process it as var[2]. The exact same short variable can be referred to as v.velocity in the robot code.

LIST OF REFERENCES

1. "PROJECT 2025 Briefing for the Secretary of the Defense and the Service Chiefs," Institute for National Strategic Studies, National Defense University, 1994

2. The Strategic Assessment Center Science Applications International Corporation, "ROBOTICS WORKSHOP 2020," June 1997.

3. Gage, D. W., "Development and Command-Control Tools for Many-Robot Systems," *Proceedings of SPIE Microrobotics and Micromechanical Systems*, Philadelphia, PA, October 1995.

4. *NOMAD 200 User"s Manual*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

5. *SCOUT Beta 1.1 Manual*, Nomadic Technologies, Inc., Mountain View, CA, 1998.

6. *NOMAD 200 Hardware Manual*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

7. *The NOMAD SCOUT Advertisement Brochure*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

8. *RangeLAN2/ISA User's Guide*, Proxim, Mountain View, CA, 1993.

9. Graham, P., *ANSI Common LISP*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1996.

10. Personal Conversation between Dr. Douglas W. Gage, Code D371, Space and Naval Warfare Systems Center San Diego, CA, and the author, 12 November, 1998.

11. Hutin, N., Pegard, C., Brassart, E., "A Communicaion Strategy for Cooperative Robots," *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems, Victoria, B.C., Canada, October 1998.*

12. Ohkawa, K., Shibata, T., Tanie, K., "Method for Generating of Global Cooperation based on Local Communication," *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems, Victoria, B.C., Canada, October 1998.*

13. Vainio, M., Pekka, A., Halme, A., "Generic Control Architecture for a Cooperative Robot System," *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems, Victoria, B.C., Canada, October 1998.*

14. Hoskins, D.A., "A Least Action Approach to Collective Behavior," *Proceedings of SPIE Microrobotics and Micromechanical Systems*, Philadelphia, PA, October 1995.

15. Stallings, W., *Data and Computer Communications*, 5th ed., Prentice Hall, Inc., Englewood Cliffs, NJ, 1996.

16. Padovano, M., *Networking Applications on Unix System V Release 4*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1993.

17. Gauthier, D., Freedman, P., Carayannis, G., Malawany, A., "Interprocess Communication for Distributed Robotics," *Multirobot Systems*, IEEE Computer Society Press, pp. 99-110, 1990.

18. SUN Microsystems, *Network Programming Guide*, Rev. A, pp. 251-344, SUN Microsystems, 1990.

19. Gomer, D., *Internetworking with TCP/IP*, v. 1, Prentice Hall, Inc., Englewood Cliffs, NJ, 1991.

20. Hiraishi, H., Ohwada, H., Mizoguchi, F., "Web-based Communication and Control for Multiagent Robots," *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems, Victoria, B.C., Canada, October 1998.*

21. Yun, X., EC 4300 Class Notes, Naval Postgraduate School, 1998

22. Kelley, A., Pohl, I., *C by Dissection: The Essentials of C Programming*, Addison-Wesley, Menlo Park, CA, 1996.

23. Winston, P., Horn, B., *LISP*, 2nd ed., Addison-Wesley, Menlo Park, CA, 1984.

24. *Language Reference Manual*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..... 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library..... 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

4. Professor Xiaoping Yun, Code EC/YX 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

5. Douglas W. Gage 2
   SPAWARSYSCEN D371
   53406 Woodward Rd.
   San Diego, California 92152-7383

6. John A. Roese (PL-TS)..... 1
   SPAWARSYSCEN D103
   53570 Silvergate Ave. Rm. 3027A
   San Diego, California 92152-5271

7. LT Uriah E. Zachary, USN 2
   2185 Johnston Rd.
   Escondido, California 92029

32 473NPG 552
TH
11/02 22527-200 NLB

DUDLEY KNOX LIBRARY
3 2768 00404634 2